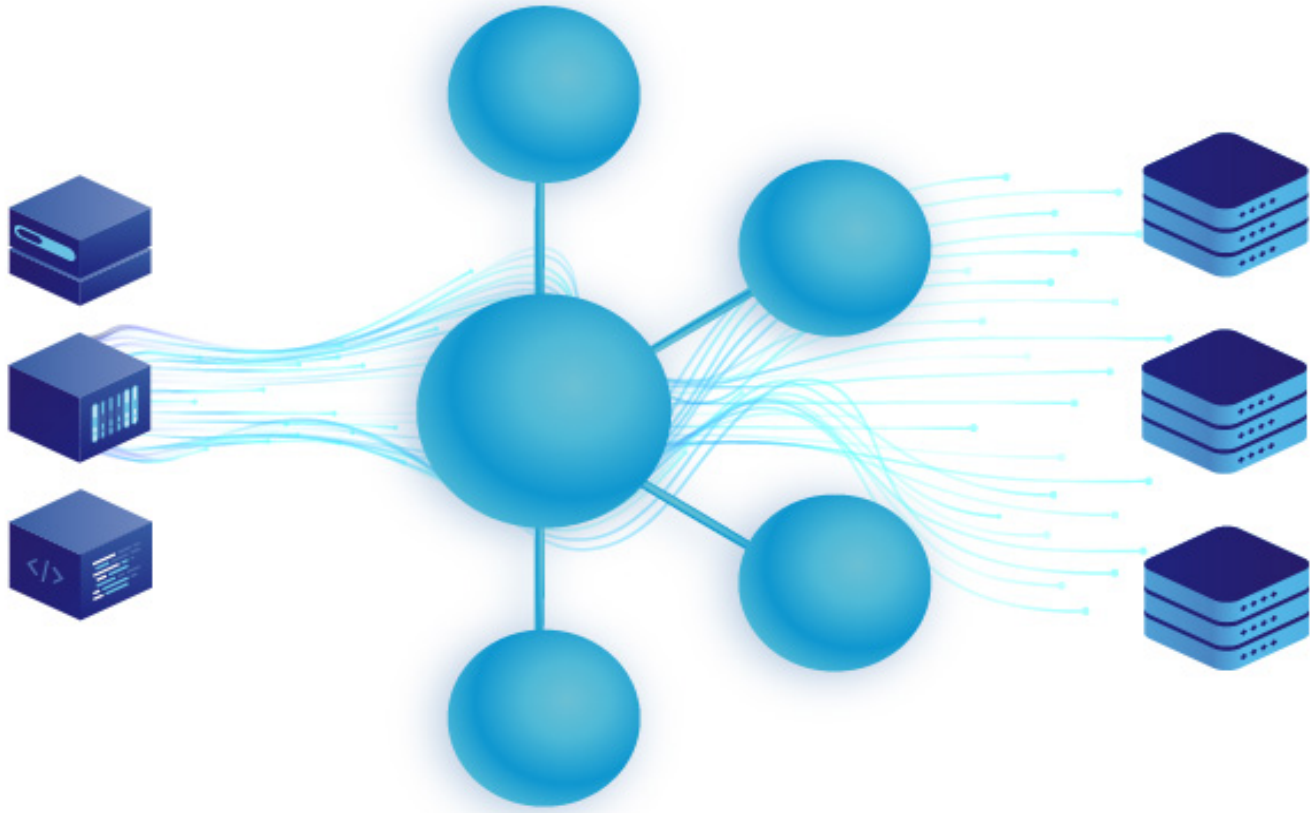
Kafka Connect e Kafka Streams

Integre suas aplicações ao Kafka usando as melhores práticas



Casa do Código | alura

RODRIGO VIEIRA PINTO

Sumário

- ISBN
- Prefácio
- Agradecimentos
- Para quem é este livro
- Sobre o livro
- Sobre o autor
- O Alura Viagens e Turismo
  - 1 Funcionalidades da solução
  - 2 Onboarding – ferramentas a serem instaladas
  - 3 Kafka Connect: Conceitos básicos
- Kafka Connect e seus recursos
  - 4 Kafka Connect e a resiliência das aplicações
  - 5 Melhorias nas mensagens postadas e o ms-pesquisa
  - 6 Busca e seleção de voos no ms-pesquisa
  - 7 Busca e seleção de hospedagens no ms-selecao-hospedagens
  - 8 Boas práticas para nomes de tópicos e outros SMTs interessantes
- Kafka Streams e algumas topologias avançadas
  - 9 Montagem de pacotes de viagem com o ms-pacote e o Kafka Streams
  - 10 Informações sobre pacotes vendidos com KTable e o ms-marketing
- Testes automatizados
  - 11 Testes automatizados de consumo de mensagens
  - 12 Testes de topologias no Kafka Streams e o RocksDB
- Encerramento
  - 13 Apêndice - Transactional Outbox Pattern (e padrões relacionados)
  - 14 Bibliografia

ISBN

Impresso: 978-85-5519-377-4

Digital: 978-85-5519-378-1

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Pinto, Rodrigo Vieira
Kafka Connect e Kafka Streams : Integre suas aplicações ao Kafka usando as melhores práticas / Rodrigo Vieira Pinto ; Vivian Matsui. -- 1. ed. -- São Paulo : Aovs Sistemas de Informática, 2024.

ISBN 978-85-5519-377-4

1. Aplicativos - Software - Desenvolvimento 2. Banco de dados - Desenvolvimento 3. Engenharia de software 4. Kafka connect (Software de aplicação) 5. Software aplicativo - Desenvolvimento 6. Software aplicativo - Estudo e ensino I. Matsui, Vivian. II. Título.

24-205947 CDD-004.2

Índices para catálogo sistemático:

1. Software : Arquitetura, projeto e desenvolvimento : Especificações técnicas : Ciência da computação 004.2

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio

Recebi com enorme felicidade o convite do Rodrigo para escrever o prefácio deste livro. É ótimo saber que teremos mais um livro sobre Kafka disponível em português. Recordo-me de que, quando comecei a trabalhar com Kafka, em 2016, não havia tanto material disponível. Na época, tive que aprender, com muita pesquisa, pouco material e muitos erros a lidar com essa ferramenta tão extraordinária.

No meu primeiro contato com Kafka, foi-me pedido para fazer uma Prova de Conceito (*Proof of Concept* – PoC) em uma "base não relacional chamada Kafka". Não sei de onde a pessoa tirou que Kafka era uma base de dados, talvez tivesse lido em algum blog lá em 2016 ou talvez tivesse assistido a uma palestra em alguma conferência... Não vou fazer pesquisa arqueológica na internet para saber onde informação surgiu, mas embora em 2018 Martin Kleppmann tenha apresentado um *keynote* no Kafka Summit com o título *Is Kafka a Database?* ("Kafka é uma base de dados?" em tradução livre), Kafka não pode ser usado como uma base de dados tradicional.

Pensando que era uma base de dados, minha primeira atividade foi criar um "*Hello World*" que enviasse uma mensagem para o *broker* Kafka e outro que lesse a mensagem. Ao ter sucesso, decidi que minha segunda atividade seria alterar essa mensagem após enviada. Pensando ser possível, passei dias tentando, li os escassos posts de blog até chegar à conclusão de que, uma vez enviada uma mensagem, ela não pode ser apagada. Nós nos acostumamos com dados mutáveis, dados que podem ser acessados e alterados, mas a maioria das aplicações Kafka, usa-se de eventos para fazer a comunicação entre os serviços. Eventos são um tipo de dado que representam "algo que aconteceu no tempo". Você não pode alterar um evento, só pode enviar um novo evento dizendo que algo novo aconteceu.

Contei essa anedota sobre meus primeiros dias com Apache Kafka para demonstrar que é preciso alinhar as nossas expectativas quando vamos aprender uma ferramenta nova. Kafka, por mais que possa atuar como uma base de dados, não é uma base de dados. Kafka, a princípio, era apenas um *Message-Oriented Middleware*, ou seja, é uma camada de software que integra sistemas heterogêneos através da troca de mensagens. Até 2015, existia o apenas o *broker* e as bibliotecas para podermos enviar e ler mensagens. Assim, as aplicações liam e escreviam mensagens de forma distribuída e independentes.

O Kafka Connect surge em 2015 (ver em https://archive.apache.org/dist/kafka/0.9.0.0/RELEASE_NOTES.html o *Release Notes da versão 0.9.0.0*) com a missão de integrar facilmente outros sistemas ao broker Kafka. Antes do Kafka Connect, se fosse para integrar uma aplicação a um broker Kafka, cada serviço deveria ser reescrito para fazer não só escrever na sua própria base de dados, mas também para enviar eventos ao *broker* Kafka. Você deve estar pensando: "*Difícil, né!*" Com toda a razão, é muito difícil integrar aplicações existentes usando apenas *producers* e *consumers* Kafka.

O Kafka Connect facilita a vida de quem precisa integrar uma aplicação existente com o Kafka. Agora não é mais preciso fazer uma escrita dupla para garantir a integridade dos dados, basta criar um *Source Connector* conforme descrito no livro e fazer com que serviços rodem independentemente. Depois de integrar sua aplicação, você pode evoluir sua arquitetura para *Event-Driven*, onde os serviços notificam uns aos outros por meio de eventos. Também será possível criar aplicações *Stream Processing*, usando o Kafka Stream, para fazer o processamento de eventos continuamente. Por fim, você pode propor na sua empresa que tratem eventos como um produto, da mesma forma que APIs são tratadas como produtos, provendo infraestrutura para documentar os tópicos e *schemas* usados.

Com sua habilidade exemplar como professor, Rodrigo conduz uma jornada dinâmica, ilustrando o uso prático do Kafka Connect na evolução de projetos, com o cenário concreto da Alura Viagens e Turismo. Este livro não apenas ensina os conceitos fundamentais do Kafka Connect, mas também capacita você a integrar seus sistemas de forma eficaz por meio dos *brokers* Kafka. Prepare-se para dominar essa ferramenta essencial com a orientação especializada de Rodrigo.

Divirta-se.

Victor Emanuel Perticarrari Osório

Agradecimentos

O trabalho de escrita de um livro dificilmente é feito sozinho. Ainda que ele possua um ou mais autores declarados na capa, diversas pessoas contribuem com a sua escrita, direta ou indiretamente. E espero aqui não deixar ninguém para trás.

Inicialmente gostaria de agradecer a diversas pessoas do PicPay, como Diego Sana, Daniel Machado Reis, Michel Nery, Eurides Baptistella, Rafael Neris e Vagner Garcia. Lá, vimos que o Kafka e o Kafka Connect possuem potencial, e pude entender melhor como essas ferramentas funcionam. E comprovar que, mesmo com ferramentas e conceitos novos, as ideias de código com qualidade e testes automatizados devem sempre estar presentes.

Também gostaria de agradecer a diversas pessoas que trabalham comigo atualmente, como Rubens Kenji Takaki Toyohara (um dos grandes amigos que fiz nessa estrada da programação), e o seu time de desenvolvedores: Daniel Sahara, Ingo Passos Souza, Nelmisson Pinheiro de Souza e Paulo Eduardo Groth, além de desenvolvedores de outros times como Leonardo Vinicius Marquezani e Jefferson Hideki Tookuni. Todos excelentes profissionais, que possuem sincera preocupação com o trabalho que desenvolvem.

Ainda no meu trabalho atual, não posso deixar de lembrar de Reginaldo Ribeiro Tostes (estou ansioso para descobrir os próximos trabalhos que faremos juntos) e de Elcio Abrahão. Devo a este último a ideia de escrita deste livro.

Obrigado também a Fernando Boaglio e a Phillip "Phil" Calçado. Aprendi e aprendo com vocês. Há anos.

Naturalmente, nem só de trabalho vive o homem. Então, esses agradecimentos não estariam completos se não mencionassem minha esposa Marcia e meu filho Leonardo. Obrigado por entenderem os momentos de concentração e isolamento. O

resultado está aqui. A vida é boa, e boa parte da culpa disso é de vocês.

Obrigado Deus! E vamos em frente!

Para quem é este livro

Este livro é indicado para desenvolvedores que já possuam experiência com a ferramenta Kafka, e que gostariam de implementar soluções resilientes, desacopladas, simples e confiáveis no processamento de mensagens assíncronas usando ferramentas como Kafka Connect e Kafka Streams. Profissionais que já tiveram que lidar com problemas de consumo ou envio de mensagens para o Kafka nas suas aplicações, independente da linguagem de programação ou biblioteca utilizadas, certamente encontrarão nessas ferramentas recursos que facilitarão bastante o seu trabalho.

Usaremos diversos bancos de dados nos exemplos, desde os relacionais como o Postgres e o MySQL até não relacionais como o MongoDB. Logo, conhecimentos relacionados a bancos de dados são importantes.

Usaremos o docker para rodá-los, nos exemplos do livro, bem como a linguagem Java nos diversos microsserviços que implementaremos ao longo dos capítulos.

Sobre o livro

Kafka é uma tecnologia que vem sendo cada vez mais adotada no mercado. Com ele, podemos criar sistemas altamente distribuídos e desacoplados, podendo transferir grandes volumes de dados com baixa latência.

Entretanto, para que as aplicações se beneficiem de suas características, é necessário que elas sejam consumidoras ou produtoras de mensagens, o que nem sempre é possível (dada a tecnologia em que a aplicação foi construída), ou mesmo desejável (produtores/consumidores podem ter sido construídos em diversas linguagens de programação, com suas respectivas bibliotecas de integração, e ainda podem precisar lidar com questões relacionadas a garantias de envio e consumo de mensagens, por exemplo).

Nesses cenários, o uso de ferramentas que atuem como intermediárias entre as aplicações e os tópicos Kafka torna-se importante, pois podemos encapsular a resolução desses problemas. E o Kafka Connect é uma delas.

Podemos encontrar bons exemplos de uso do Kafka Connect em aplicações mais antigas, onde as tecnologias usadas não suportam a construção de produtores de mensagens Kafka. E, graças a conectores previamente instalados no Kafka Connect, podemos conectar diversas fontes de dados dessas aplicações a tópicos Kafka, sem que sejam necessárias grandes alterações nas aplicações.

Felizmente, seu uso não se restringe somente a sistemas legados. A extração de partes de código de um monolito para a construção de microsserviços também pode se beneficiar dessa tecnologia. O monolito pode, por exemplo, compartilhar dados de suas fontes de dados com os microsserviços extraídos de forma transparente.

O Kafka Connect pode ser considerado, portanto, uma central de dados, onde diversas fontes (sejam elas bases relacionais, arquivos, filas ou sistemas como SAP, Service Now, Salesforce) podem disponibilizar a ele suas informações de forma simples e sem a necessidade de escrita de código.

Ao lado do Kafka Connect, temos também a tecnologia Kafka Streams. Com ela, é possível tratar o recebimento de mensagens como eventos, de um ou mais tópicos, realizando desde operações semelhantes a que realizamos com tabelas em bancos de dados, como junções (*joins*), uniões (*unions*) ou até mesmo operações encontradas nas *collections* do Java como *filter* ou *map*.

Para o estudo dessas ferramentas, usaremos um estudo de caso fictício envolvendo a compra de passagens aéreas e hospedagens, o Alura Viagens e Turismo. Ele será composto de diversos microsserviços, que serão apresentados ao longo dos capítulos. Veremos diversas maneiras de conectá-los ao Kafka usando o Kafka Connect e bancos relacionais e não relacionais como fontes de dados. Estudaremos alguns conceitos relacionados a resiliência de aplicações, e veremos como o Kafka Connect nos ajuda com isso. Implementaremos manipulações nos dados diretamente no Kafka Connect usando *Single Message Transformations* (SMTs) e transformações de dados usando dados de diversos tópicos usando o Kafka Streams. Também escreveremos testes automatizados que mostrem que o código que criamos está funcionando corretamente. Por último, no apêndice, implementaremos padrões como *Transactional Outbox* nos microsserviços, e utilizaremos o Kafka Connect como parte da implementação.

Sobre o autor

Rodrigo Vieira Pinto é Mestre em Engenharia de Software pelo IPT-SP (Instituto de Pesquisas Tecnológicas de São Paulo) e Pós-graduado em Engenharia de Software pela PUC-SP. Possui cerca de 25 anos de experiência em desenvolvimento de sistemas em Java, PHP e Ruby, passando também por C. Já trabalhou com servidores de aplicação como JBoss e Websphere, usando frameworks como EJB 2 e 3, JSP/Servlet, JSF e Hibernate (e ficou feliz quando o Spring Boot surgiu). Gosta muito de paradigmas de programação, padrões de projeto e de testes automatizados.

Também ministra aulas de programação desde 2018, tanto em cursos livres quanto em faculdades, passando por Caelum, Alura, São Paulo Tech School (SPTech) e FIAP.

Nas horas vagas, estuda tecnologias alternativas, história do Brasil e religião, além de praticar esportes como natação e corrida e assistir corridas de automóvel.

Amante de software bem escrito, entende que a TI tem muito ainda a crescer e evoluir. E a gente, muito a contribuir.

O Alura Viagens e Turismo

Seja bem-vindo(a) ao Alura Viagens e Turismo. Aqui o viajante está sempre em primeiro lugar!

Usaremos uma arquitetura baseada em microsserviços, cujas comunicações serão realizadas de forma assíncrona. Para isso, usaremos o Kafka e ferramentas como o Kafka Connect e o Kafka Streams.

Embarque nesta viagem com a gente!

Capítulo 1
Funcionalidades da solução

Alura Viagens e Turismo é uma *startup* que possui objetivos de se tornar uma das maiores empresas do ramo. Para isso, seus desenvolvedores construíram diversos microsserviços, além de utilizarem soluções baseadas em Kafka, como o Kafka Connect e o Kafka Streams.

De forma semelhante a diversas soluções do mesmo segmento, o Alura Viagens e Turismo possuirá funcionalidades como:

- busca de voos nas companhias aéreas;
- serviço de pesquisa e reserva de voos;
- serviço de pesquisa e reserva de hospedagens;
- geração de pacotes de viagem, com base nos voos e hospedagens selecionadas;
- extração de dados de pacotes criados por hospedagem, para ações de marketing.

Toda a interação com os possíveis usuários finais é feita usando APIs REST ou tópicos Kafka, de forma a simplificar a solução. Assim, não usaremos interfaces gráficas *web*, *desktop* ou mesmo *mobile*.

Para entendermos melhor a arquitetura do Alura Viagens e Turismo, tomaremos contato com a solução incrementalmente. Em cada capítulo, veremos inicialmente alguns diagramas que nos mostrarão a modelagem da solução. Em seguida, veremos o código que implementa efetivamente as funcionalidades modeladas.

Capítulo 2
Onboarding – ferramentas a serem instaladas

Para que possamos aprender melhor com os exemplos deste livro, é fundamental que as seguintes ferramentas sejam instaladas no seu computador:

- JDK 21 ou superior
- A IDE Java de sua preferência (Eclipse, IntelliJ, Visual Studio Code etc.)
- Maven (versão 3.9.6 ou mais recente)
- Um *client* de banco de dados de sua preferência (Squirrel, DBeaver ou outros para bases relacionais) e um para o MongoDB, como o Studio 3T. Ou mesmo o DbSchema, que oferece suporte a todas as bases de dados usadas nos exemplos deste livro.
- git

O Docker também será usado para rodarmos o Kafka, o Kafka Connect e os bancos de dados MySQL, MongoDB e Postgres. Entretanto, sinta-se à vontade para instalar esses softwares diretamente na sua máquina. De qualquer forma, os exemplos ao longo do livro foram escritos com essas ferramentas rodando em contêineres docker. À medida que as ferramentas forem introduzidas nos capítulos, será mostrado o seu código, presente também no `docker-compose.yml` da solução. Os microsserviços Java são rodados na IDE ou via linha de comando.

O código-fonte do projeto encontra-se em https://github.com/rodrigovp/livro-kafka-connect, e todos os capítulos estão nas *branches* do projeto. A *branch* **main** contém o início do projeto. O conteúdo visto no **capítulo 3** está na *branch* cap-3, a *branch* **cap-4** contém o conteúdo do **capítulo 4** e assim por diante.

Há também um projeto que pode ser importado no Postman, o `livro-kafka.postman_collection.json`, para que possamos testar as chamadas REST às nossas aplicações. Mas sinta-se à vontade para usar o *client* REST de sua preferência.

Com o ambiente preparado, vamos começar.

Capítulo 3
Kafka Connect: Conceitos básicos

O Alura Viagens e Turismo possui convênios com diversas companhias aéreas. Assim, conseguimos sempre ter diversas opções de horários e preços em nossas pesquisas. As companhias aéreas preparam seus voos e os cadastram em suas aplicações, que serão representadas na solução pelo *ms-companhias-aereas*.

É importante que o Alura Viagens saiba o quanto antes da existência de todos esses voos, para que eles possam ser disponibilizados a todos os viajantes. Logo, precisamos integrar as aplicações, e precisamos também garantir que todos os voos sejam enviados assim que forem criados.

Os voos serão recebidos pelo *ms-pesquisa*, um dos microsserviços do Alura Viagens e ponto de entrada dos voos na empresa. Há, portanto, uma integração entre o *ms-companhias-aereas* e o *ms-pesquisa*. É possível realizarmos essa integração de algumas maneiras:

- o *ms-pesquisa* pode disponibilizar uma API REST que recebe os novos voos. Neste caso, o *ms-companhias-aereas* envia os novos voos usando o protocolo REST, e espera uma resposta do *ms-pesquisa* de que os voos foram recebidos com sucesso.

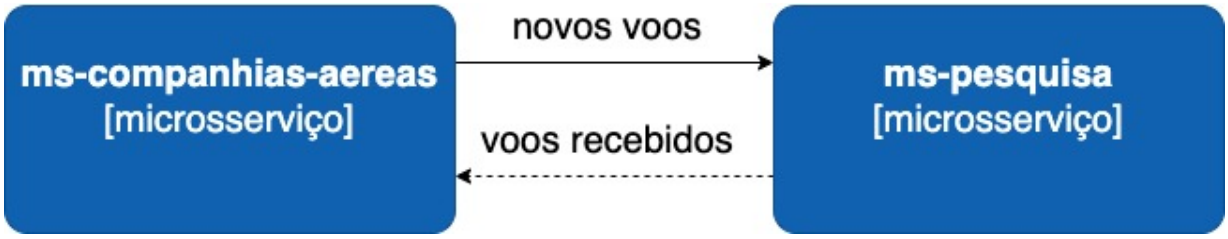


Figura 3.1: Integração das aplicações usando REST.

- a infraestrutura da Alura Viagens pode disponibilizar uma fila de mensagens, que pode ser alimentada pelas companhias aéreas.

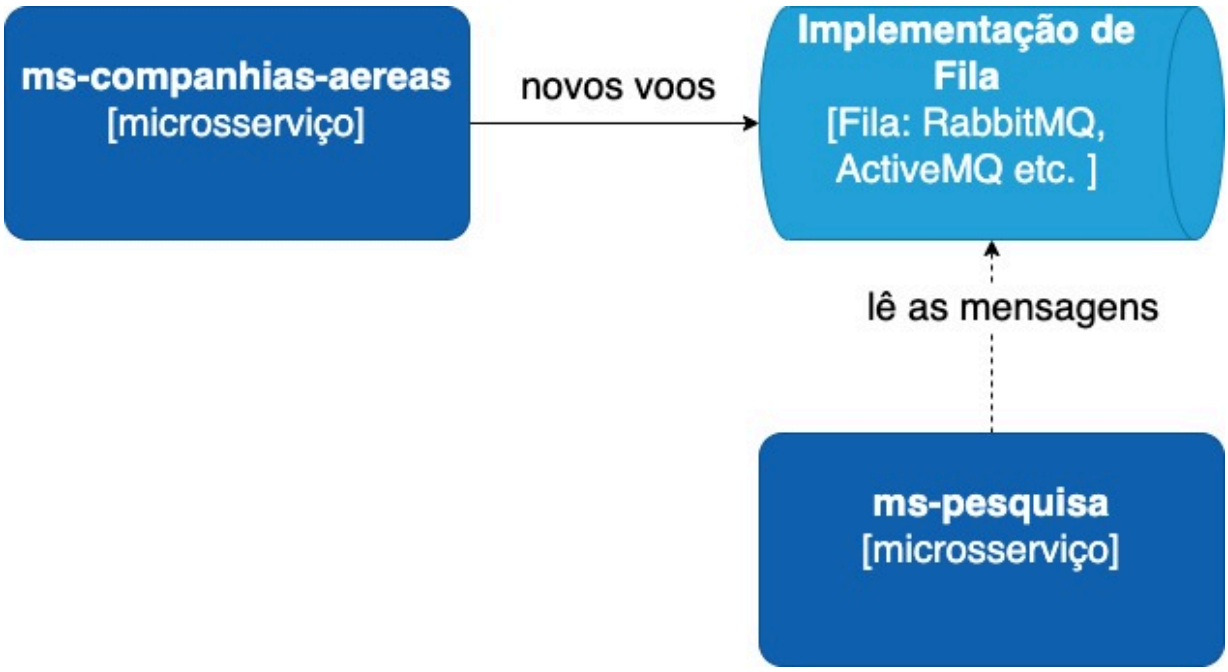


Figura 3.2: Integração das aplicações usando fila.

Essas soluções são passíveis de serem aplicadas, embora possuam algumas desvantagens:

- as instâncias do *ms-pesquisa* podem, mesmo que por alguns instantes, ficar fora do ar, e as instâncias do *ms-companhias-aereas* podem não estar preparadas para uma situação dessas. Consequentemente, o *ms-companhias-aereas* pode realizar chamadas à API REST e não receber nenhuma resposta, fazendo com que ocorram erros. Há também situações em que as instâncias do *ms-pesquisa* estão no ar, mas, por diversas razões, podem demorar a responder, causando também lentidão no processamento realizado pelas instâncias do *ms-companhias-aereas*. Por último, serão necessários controles nos contratos da API. Caso ocorra uma atualização de contrato, todas as instâncias do *ms-companhias-aereas* precisarão ser atualizadas.

- ao optarmos por inserir uma fila entre as aplicações, fazemos com que o processo se torne assíncrono. Assim, o *ms-companhias-aereas* não precisa esperar respostas do *ms-pesquisa*, o que pode ser muito bom. Entretanto, podemos

perder a oportunidade de paralelizar a leitura das mensagens por diversas aplicações além do *ms-pesquisa*, visto que, em uma fila básica, dada 1 mensagem, apenas 1 aplicação conseguirá lê-la e processá-la.

Parece que a solução mais adequada é tornar essa integração assíncrona, mas não com filas, e sim com tópicos.

Podemos então substituir a implementação de fila por um software que lide com tópicos. Por exemplo, o Kafka.

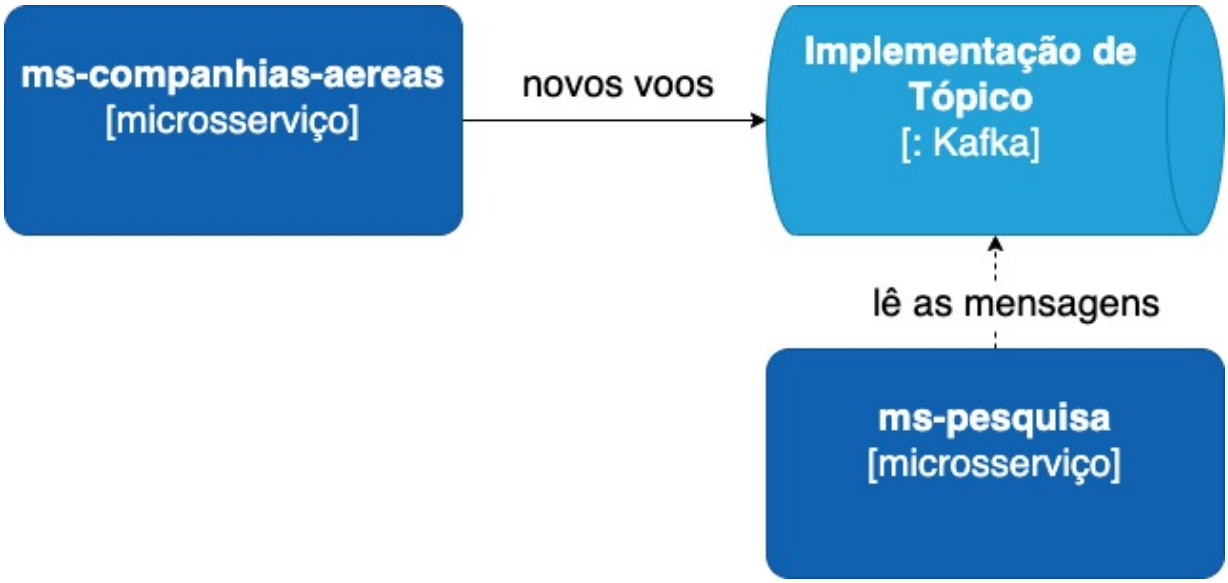


Figura 3.3: Integração das aplicações usando Kafka.

Com essa arquitetura, tornamos o processamento de mensagens assíncrono, além de ser possível integrar diversos consumidores ao tópico presente no Kafka, se necessário. As empresas aéreas podem, portanto, enviar seus voos para a Alura Viagens, bastando enviar mensagens para o tópico apropriado, o que envolveria a conexão com o Kafka da Alura Viagens.

Para que isso seja possível, é necessário que o *ms-companhias-aereas* realize tanto a persistência dos voos em banco de dados quanto o envio para o tópico. A arquitetura da solução ficaria como na figura a seguir:

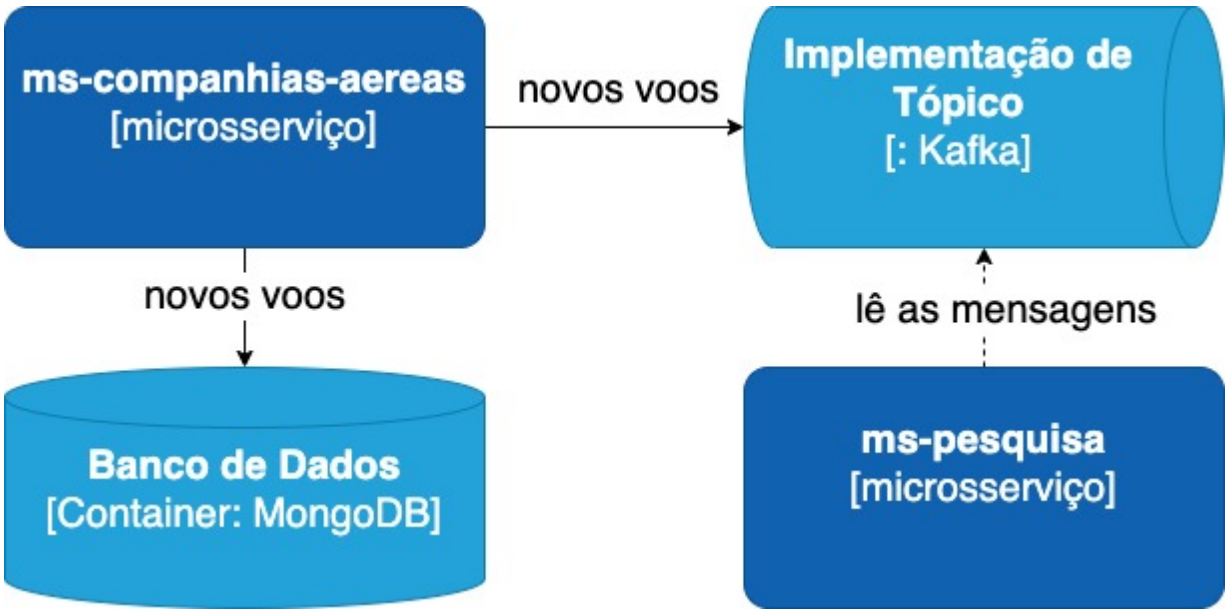


Figura 3.4: Envio dos voos tanto para o MongoDB quanto para o Kafka.

Note que o MongoDB foi o banco de dados escolhido para o *ms-companhias-aereas*. Ele está configurado no `docker-compose.yml` da maneira a seguir:

```
    mongodb:
        image: mongo:4.2
        hostname: mongodb
        container_name: mongodb
        ports:
            - 27017:27017
        environment:
            MONGO_INITDB_DATABASE: livro-kafka
            MONGO_INITDB_ROOT_USERNAME: user
            MONGO_INITDB_ROOT_PASSWORD: password
        volumes:
            - ./scripts/init-mongodb.js:/docker-entrypoint-initdb.d/init-
mongodb.js:ro
        command: mongod --replSet rs0 --auth
        healthcheck:
            test: test $$(echo "rs.initiate().ok || rs.status().ok" |
mongo -u user -p password --quiet) -eq 1
            interval: 10s
```

```
            start_period: 30s
    networks:
        bubble:
```

> Repare que o MongoDB está rodando como *replica set*. No capítulo 4 veremos os motivos disso.

Mas como garantir que as instâncias de *ms-companhias-aereas* farão a persistência dos voos nas suas bases *e* o envio para o tópico? Se uma das 2 operações falhar, a outra não pode acontecer.

> Estamos falando apenas da comunicação entre dois microsserviços. Se houvesse mais, lidaríamos com diversos pontos de falha, não apenas um. Há muitas maneiras de um sistema falhar, mesmo com o código perfeitamente organizado e testado.
>
> Sam Newman, autor dos livros *Building Microservices* (2021) e *Monolith to Microservices* (2019) e um dos primeiros autores a escrever livros sobre microsserviços, escreveu essa frase em uma de suas redes sociais (em tradução livre):
>
> *"Cheguei à conclusão de que existem duas regras de ouro da computação distribuída:*
>
> 1. *Você não pode transmitir informações do ponto A ao ponto B instantaneamente, e*
> 2. *Às vezes, a coisa (ou o sistema) com quem você quer conversar não está lá. Todos os outros desafios dos sistemas distribuídos decorrem destas regras."*

Talvez a segunda regra de ouro é a que mais se aplique ao nosso cenário. Como garantir que os voos sejam enviados para todos os sistemas com consistência e garantias?

Podemos pensar em alguns controles complexos entre as classes do *ms-companhias-aereas* que lidam com o banco e as que lidam com o Kafka, como aguardar o banco de dados confirmar a transação de persistência dos voos, e em seguida realizar o envio deles para o Kafka. E, caso o envio falhar, realizar um *rollback* no banco. Ou mesmo enviar os voos ao Kafka e, somente depois que o Kafka confirmar que os recebeu, enviá-los ao banco de dados. Mas podemos tambem pensar em outras soluções.

3.1 O Kafka Connect

Essas integrações poderiam também ser realizadas utilizando aplicações específicas para isso, usando diversas linguagens de programação. Entretanto, teríamos que nos preocupar com questões relacionadas à transação, garantias de envio de mensagens, entre outras. Com o Kafka Connect, tudo o que precisamos é de um *connector plugin* e de um conector. O Kafka Connect cuida do resto.

O Kafka Connect, de forma bem sucinta, é uma ferramenta que permite:

- o envio de dados de uma fonte de dados para tópicos Kafka;
- o envio de dados de um tópico Kafka para uma fonte de dados.

Para isso:

- instalamos um *connector plugin* com suporte à fonte de dados;
- configuramos um conector que acesse a fonte de dados;
- instalamos esse conector no Kafka Connect.

Por "fonte de dados", podemos pensar em bancos de dados relacionais ou não relacionais, sistemas de log como Elasticsearch, ERPs como SAP, repositórios de buckets como AWS S3 e Google Cloud Storage e até sensores e dispositivos IoT. Todas as manipulações realizadas nas fontes de dados podem ser captadas

pelo Kafka Connect, sejam elas de inclusão, alteração e remoção de dados. Essas manipulações são tratadas como **eventos** pelo Kafka Connect.

Connector plugins

A instalação básica do Kafka Connect não nos permite conectar com nenhuma fonte de dados. É necessário que instalemos antes o *connector plugin* adequado à fonte que desejamos acessar.

> Esse é um conceito muito semelhante à conexão com bancos de dados usando Java/JDBC. Os *connector plugins* são, portanto, equivalentes aos drivers JDBC, com a diferença de que os *connector plugins* são usados para todas as fontes de dados, inclusive aquelas que não são bancos de dados.

Os *connector plugins* podem ser de dois tipos: **source** e **sink**:

- os *connector plugins* **source** leem dados das fontes de dados e os enviam para o Kafka;
- os *connector plugins* **sink** fazem o contrário dos conectores **source**. Ou seja, leem dados de tópicos Kafka e os enviam para fontes de dados;
- ambos: são tanto **source** quanto **sink**.

A seguir, uma figura que ilustra a relação entre os conectores, as fontes de dados e o Kafka Connect:

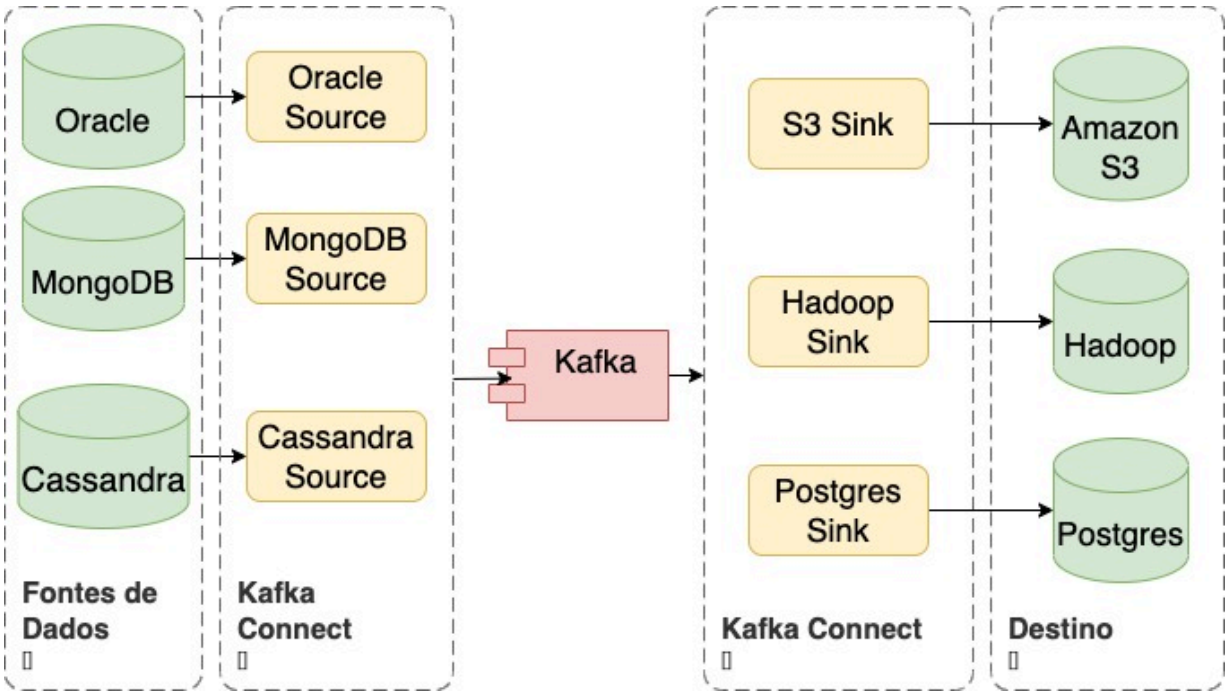


Figura 3.5: Relação entre os connector plugins, as fontes de dados e o Kafka Connect.

Existem diversos conectores, para diferentes fontes de dados, produzidos por diversos fabricantes. Neste capítulo, usaremos o conector para MongoDB do projeto Debezium.

> Os *connector plugins* da Debezium são recomendados inclusive pela Confluent, a empresa que desenvolveu o Kafka e o entregou ao projeto apache. A Confluent também possui os seus próprios conectores. Uma relação dos *connector plugins* Debezium encontra-se em https://debezium.io/documentation/reference/stable/connectors/index.html. E a relação dos conectores fornecidos pela Confluent, em https://www.confluent.io/product/connectors/

Conectores

Os conectores consistem em um conjunto de informações do tipo chave-valor, escritos em formato JSON. Esse conjunto de informações deve ser, basicamente:

- as configurações de acesso à fonte de dados e o tópico (no caso de conectores do tipo **source**);
- as configurações do tópico e a fonte de dados de destino (para conectores **sink**);
- alguns ajustes finos, que podem estar relacionados a quais dados precisarão ser lidos do banco, transformadores de mensagens, entre outros;
- configurações exclusivas do conector que se deseja usar.

A figura a seguir nos mostra alguns exemplos de conectores **source** conectando-se a bancos de dados de diversos tipos e enviando os registros lidos para tópicos. Note a necessidade do *connector plugin*, selecionado de acordo com a fonte de dados. O Kafka Connect atua como um *listener* de fontes de dados e, à medida que ocorre alguma operação de manipulação de dados nas fontes, ele envia mensagens para os tópicos contendo as manipulações realizadas.

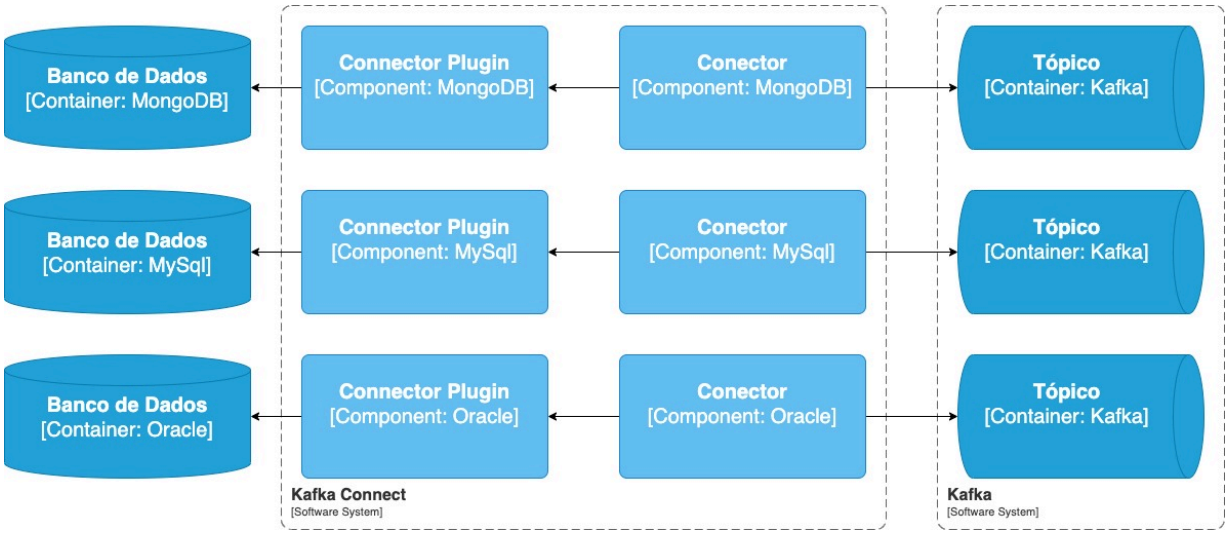


Figura 3.6: Relação entre os connector plugins e os conectores.

Neste capítulo, veremos como conectar uma fonte de dados MongoDB (que é o banco do *ms-companhias-aereas*) a um tópico Kafka. Ao longo do livro, veremos também a criação de outros conectores, bem como a criação de transformadores.

3.2 Criando nossa imagem de Kafka Connect

O Kafka Connect pode ser instalado diretamente na máquina do desenvolvedor, ou podemos executá-lo como um contêiner docker.

Uma opção de criação de imagem docker é criando-a com os *connector plugins* e todas as demais ferramentas prontas para uso. Mas é possível também instalar os *connector plugins* mais tarde. A primeira opção é interessante, pois nos permite ver detalhes dos *connector plugins*, bem como preparar uma imagem docker de Kafka Connect totalmente customizada para as nossas necessidades.

A instalação dos *connector plugins* deve obedecer a algumas regras:

- os arquivos relacionados devem estar em uma pasta definida pela variável de ambiente `KAFKA_CONNECT_PLUGINS_DIR`, que é definida no momento da instalação do software;
- é possível criarmos quantas pastas quisermos, a partir da pasta especificada pela variável acima. Isso nos permite organizar nossos *connector plugins* segundo algum critério.

Sendo assim, para o nosso projeto, vamos usar um *Dockerfile* que contém:

- a definição da variável de ambiente;
- instruções para download de todos os *connector plugins* de que precisamos para toda a solução.

O `Dockerfile` foi construído passo a passo, com mensagens que indicam o que está sendo feito no momento, visando facilitar o entendimento de sua estrutura. Usaremos como imagem base uma imagem (https://hub.docker.com/r/confluentinc/cp-kafka-connect-base) da empresa Confluent (https://www.confluent.io).

> O projeto Debezium oferece imagens com todos os connector plugins, o que pode facilitar o desenvolvimento para quem está começando com o Kafka Connect. Entretanto, é importante conhecer como o Kafka Connect pode ser customizado e preparado conforme a sua necessidade. Por isso, os exemplos deste livro usarão a imagem da Confluent.

```
FROM confluentinc/cp-kafka-connect-base:7.4.1

ENV KAFKA_CONNECT_PLUGINS_DIR=/etc/confluent/connect
RUN mkdir -p $KAFKA_CONNECT_PLUGINS_DIR

RUN echo ">>>>>>>>>> Instalando Plugin do mongodb <<<<<<<<<<"
RUN wget  https://repo1.maven.org/maven2/io/debezium/debezium-connector-
mongodb/2.4.0.Final/debezium-connector-mongodb-2.4.0.Final-plugin.tar.gz
RUN tar xvfz debezium-connector-mongodb-2.4.0.Final-plugin.tar.gz --
directory $KAFKA_CONNECT_PLUGINS_DIR

RUN echo ">>>>>>>>>> Removendo arquivos de instalação dos conectores
<<<<<<<<<"
RUN rm *.gz

# RUN /etc/confluent/docker/run &
```

> Ao longo do livro, vamos revisitar esse `Dockerfile` à medida que a solução for apresentada. Vamos ver outros *connector plugins*, entre outras ferramentas suportadas pelo Kafka Connect.

Repare que uma das primeiras linhas de código do `Dockerfile` contêm instruções de download do *connector plugin* do MongoDB. O *ms-companhias-aereas* usa o MongoDB como base de dados, logo, precisaremos desse *connector plugin*. Ele contém diversas bibliotecas empacotadas em um arquivo *tar.gz*. Assim que o download é realizado, é feita instalação, desempacotando-o na pasta especificada pelo `KAFKA_CONNECT_PLUGINS_DIR` .

```
RUN echo ">>>>>>>>>> Instalando Plugin do mongodb <<<<<<<<<<"
RUN wget  https://repo1.maven.org/maven2/io/debezium/debezium-connector-mongodb/2.3.0.Final/debezium-connector-mongodb-2.3.0.Final-plugin.tar.gz
RUN tar xvfz debezium-connector-mongodb-2.3.0.Final-plugin.tar.gz --directory $KAFKA_CONNECT_PLUGINS_DIR
```

Com o `Dockerfile` devidamente escrito, podemos criar a imagem docker executando o script de criação de nossa imagem, presente em `docker/cdc-kafka-connect` :

```
./dockerizar-kafka-connect.sh
```

Com a nossa imagem preparada, vamos partir para os primeiros microsserviços implementados.

3.3 Integração entre o ms-companhias-aereas e o ms-pesquisa

> O *ms-pesquisa* será implementado nos capítulos 5 e 6. Neste capítulo, vamos implementar toda a infraestrutura necessária para que ele receba os voos sem problemas.

Para termos garantias de que o tanto o banco de dados quanto o Kafka recebam os novos voos, vamos inserir o Kafka Connect na arquitetura da solução, modificando nosso diagrama atual:

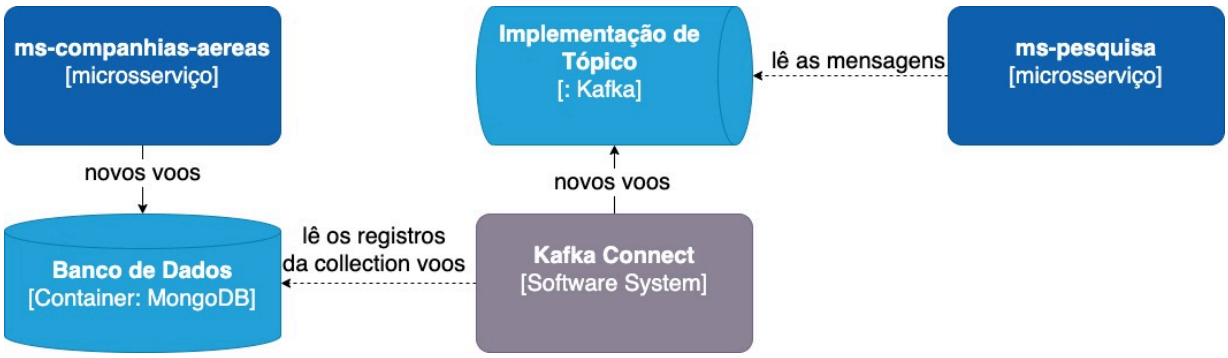


Figura 3.7: Integração das aplicações usando REST.

Para rodarmos o Kafka Connect, usaremos a imagem criada no tópico anterior, referenciando-a no nosso `docker-compose.yml`, presente na pasta `docker` do projeto.

O Kafka Connect possui muitas outras variáveis de ambiente obrigatórias, além do `KAFKA_CONNECT_PLUGINS_DIR`, e outras que são opcionais. E há ainda as variáveis que precisaremos usar devido a algumas particularidades do nosso ambiente. Vamos ver cada uma delas:

- `CONNECT_BOOTSTRAP_SERVERS`: contém todas as instalações do cluster Kafka que receberá as mensagens. No nosso caso, usaremos apenas 1 instância;
- `CONNECT_GROUP_ID`: o grupo a que o Kafka Connect vai pertencer. Importante quando precisamos lidar com um cluster de Kafka Connect;
- `CONNECT_REST_PORT`: a porta do Kafka Connect;
- `CONNECT_REST_ADVERTISED_HOST_NAME`: é o host que deve ser usado pelos demais contêineres. Veremos mais à frente que o *client* que usaremos para acessar o Kafka e o Kafka Connect precisará dessa informação.

Todas as variáveis começam com o prefixo `CONNECT_` devido ao fato de a imagem que estamos usando ser da Confluent. Outras imagens, como a do projeto Debezium, não possuem essa exigência, podendo mesmo possuir variáveis com os mesmos nomes, mas sem esse prefixo.

As três próximas configurações dizem respeito a tópicos de controle do Kafka Connect. Assim que ele voltar ao ar após alguma interrupção, esses tópicos serão lidos por ele, para que ele possa retornar seu trabalho de onde parou, bem como saber quais conectores foram configurados, além de outras informações relevantes. Os valores inseridos no `docker-compose.yml` são os normalmente usados em diversas instalações.

- `CONNECT_CONFIG_STORAGE_TOPIC` : informa o tópico onde os conectores serão armazenados quando instalados no Kafka Connect;
- `CONNECT_OFFSET_STORAGE_TOPIC` : informa o tópico onde devem ser armazenados os offsets de consumo de mensagens;
- `CONNECT_STATUS_STORAGE_TOPIC` : informa o tópico onde devem ser armazenados os *status* dos conectores.

As três configurações seguintes também dizem respeito aos tópicos de controle, mas não são obrigatórias. Foram inseridas no `docker-compose.yml` devido aos seus valores *default* serem diferentes do que precisamos. Seus valores são relacionados às instâncias de Kafka a que o Kafka Connect vai conectar. O valor padrão para todas elas é 3, mas, no nosso caso, usaremos 1 devido a rodarmos apenas uma instância de Kafka na nossa solução.

- `CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR` : diz respeito a quantas réplicas dos dados de configuração serão mantidas;
- `CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR` : idem ao acima, mas para armazenamento do offset dos tópicos;

- `CONNECT_STATUS_STORAGE_REPLICATION_FACTOR` : semelhante aos anteriores, mas configura a replicação dos dados relacionados ao status dos conectores no Kafka Connect.

> Embora os tópicos acima ajudem no controle de mensagens já processadas e outros, é possível que, caso ocorra um *restart* da aplicação, sejam enviadas aos tópicos mensagens já enviadas anteriormente. Por isso, é importante que os consumidores dos tópicos implementem padrões que evitem que mensagens duplicadas sejam reprocessadas, como o **Idempotent Consumer** (https://microservices.io/patterns/communication-style/idempotent-consumer.html).
>
> A documentação do Debezium fornece mais detalhes para cada *connector plugin* fornecido por eles. Acesse https://debezium.io/documentation/reference/stable/connectors/index.html para mais informações.

- `CONNECT_KEY_CONVERTER` : para especificar o conversor da chave da mensagem a ser usada nos conectores;
- `CONNECT_VALUE_CONVERTER` : idem ao acima, mas para o valor da mensagem;
- `CONNECT_PLUGIN_PATH` : a pasta raiz de todos os *connector plugins*. É o mesmo valor utilizado na variável `KAFKA_CONNECT_PLUGINS_DIR` .

Por último, foram inseridos comandos que verificam se o Kafka está no ar, bem como se o Kafka Connect subiu sua infraestrutura.

A configuração completa do Kafka Connect no `docker-compose.yml` está adiante:

```
kafka-connect:
    image: cdc-kafka-connect
    hostname: kafka-connect
    container_name: kafka-connect
    depends_on:
        - kafka
```

```
        - mongodb
    ports:
        - '8083:8083'
    environment:
        CONNECT_BOOTSTRAP_SERVERS: kafka:29092
        CONNECT_GROUP_ID: kafka-connect
        CONNECT_REST_PORT: 8083
        CONNECT_REST_ADVERTISED_HOST_NAME: "kafka-connect"
        CONNECT_CONFIG_STORAGE_TOPIC: _connect-configs
        CONNECT_OFFSET_STORAGE_TOPIC: _connect-offsets
        CONNECT_STATUS_STORAGE_TOPIC: _connect-status
        CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: "1"
        CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: "1"
        CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: "1"
        CONNECT_KEY_CONVERTER:
org.apache.kafka.connect.storage.StringConverter
        CONNECT_VALUE_CONVERTER:
org.apache.kafka.connect.json.JsonConverter
        CONNECT_PLUGIN_PATH: /etc/confluent/connect
    volumes:
        - $PWD/data:/data
        - ./secrets/:/etc/confluent/connect/secrets:ro
# In the command section, $ are replaced with $$ to avoid the error
'Invalid interpolation format for "command" option'
    command:
        - bash
        - -c
        - |
          echo ">>>>>>>> Launching Kafka Connect worker"
          /etc/confluent/docker/run &
          #
          echo "Waiting for Kafka Connect to start listening on
$$CONNECT_REST_ADVERTISED_HOST_NAME ⏳"
          while [ $$(curl -s -o /dev/null -w %{http_code}
http://$$CONNECT_REST_ADVERTISED_HOST_NAME:$$CONNECT_REST_PORT/connectors)
-ne 200 ] ; do
            echo -e $$(date) " Kafka Connect listener HTTP state: "
$$(curl -s -o /dev/null -w %{http_code}
http://$$CONNECT_REST_ADVERTISED_HOST_NAME:$$CONNECT_REST_PORT/connectors)
```

```
" (waiting for 200)"
           sleep 1
         done
         #
         sleep infinity
    networks:
        bubble:
```

Também precisamos da configuração do Kafka para o utilizarmos na nossa solução:

```
kafka:
    image: confluentinc/cp-server:7.3.3
    hostname: kafka
    container_name: kafka
    depends_on:
        - zookeeper
    ports:
        - "9092:9092"
        - "29092:9092"
    environment:
        KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
        KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
        KAFKA_CONFIG_STORAGE_REPLICATION_FACTOR: 1
        KAFKA_OFFSET_STORAGE_REPLICATION_FACTOR: 1
        KAFKA_STATUS_STORAGE_REPLICATION_FACTOR: 1
        KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1

        KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'false'
        KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
        KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    networks:
        bubble:
```

Repare que:

- como utilizaremos apenas 1 instância de Kafka, precisamos configurá-lo para que os tópicos possuam fator de replicação igual a 1 (são as configurações que terminam com o sufixo `_REPLICATION_FACTOR`);
- ele foi configurado para não criar os tópicos automaticamente. Logo, usaremos também um contêiner que apenas crie os tópicos necessários e encerre logo em seguida:

```
criador-topicos:
    image: confluentinc/cp-kafka:7.3.2
    container_name: criador-topicos
    depends_on:
        - kafka
    restart: on-failure
    command:
        - bash
        - -c
        - |
          echo -e 'Aguardando Kafka subir com sucesso...'
          kafka-topics --bootstrap-server kafka:29092 --list

          kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic ms-companhias-aereas.livro-
kafka.voos

          echo -e 'Criados os seguintes topicos com sucesso:'
          kafka-topics --bootstrap-server kafka:29092 --list
    networks:
        bubble:
```

É uma boa prática criar os tópicos no Kafka antes de seu uso, em detrimento de sua criação automática por parte dos produtores de mensagens. Dessa forma, facilita-se a gestão dos tópicos na ferramenta, bem como sua manutenção.

3.4 O conector para a base de dados do ms-companhias-aereas

Conforme falado anteriormente, os conectores são arquivos JSON simples que indicam sua fonte de dados e o destino. Esses conectores são armazenados nos tópicos de controle do Kafka Connect.

O JSON segue o seguinte formato:

```
{
    "name": "valor"
    "config": {
        "configuracao": "valor",
        "outra_configuracao": "outro_valor"
   }
}
```

Como se trata de uma base de dados MongoDB, a configuração que usaremos inicialmente no conector são as seguintes:

- `"name": "novos-voos-conector"` é o identificador do conector. No caso, `novos-voos-conector` ;

A seguir, as informações presentes no elemento `config` :

- `"connector.class": "io.debezium.connector.mongodb.MongoDbConnector"` : a classe de conexão com o banco de dados. Essa é uma das configurações que possuem semelhança com as utilizadas quando se utiliza a tecnologia JDBC de acesso ao banco;
- `"mongodb.connection.string": "mongodb://mongodb:27017",` : a URL de conexão com o banco de dados;
- `"mongodb.user": "user"` : o usuário de conexão com o MongoDB. Conforme o `docker-compose.yml` , o usuário é *user*;
- `"mongodb.password": "password"` : a senha do usuário acima;
- `"database.include.list": "livro-kafka"` : o usuário utilizado pode ter acesso a diversos bancos de dados. Essa configuração indica

quais bancos o conector deve escutar. Caso sejam usados mais bancos de dados, separá-los usando vírgula;

- `"collection.include.list": "livro-kafka.voos"` : aqui configuramos quais *collections* serão escutadas por esse conector. Assim como a configuração acima, caso o conector precise acessar mais *collections*, pode-se utilizar vírgula como separador;
- `"key.converter": "org.apache.kafka.connect.json.JsonConverter"` : o conversor a ser usado para o valor contido na chave da mensagem;
- `"key.converter.schemas.enable": "false"` : aqui informamos se queremos ou não que seja enviado também o *JSON Schema* associado à chave da mensagem.;
- `"skipped.operations": "u,d,t"` : toda operação que altere o estado da(s) *collection(s)* configuradas será ouvida pelo Kafka Connect, por padrão. Essa configuração informa quais operações **não** devem ser ouvidas. A seguir, as opções disponíveis:
  - `u` : para não ouvir atualizações (*updates*) nos dados;
  - `d` : para não ouvir remoções (*deletes*) dos dados;
  - `t` : para não ouvir remoções usando o `TRUNCATE` do banco de dados. Por padrão, essa configuração é `t`. Entretanto, o MongoDB não possui uma operação nativa de `TRUNCATE`. Logo, no caso desse banco de dados, o padrão é equivalente a ouvir todas as operações de alteração de dados. Logo, a única operação que será escutada pelo Kafka Connect é a de inserção de dados na *collection*.
- `"topic.prefix": "ms-companhias-aereas"` : um nome que identifique a fonte de dados para o Kafka Connect. O nome a ser dado é de livre escolha. Então, colocamos o nome do microsserviço que envia novas informações para a fonte de dados.

Com as configurações acima, nosso conector com a base do *ms-companhias-aereas* ficará assim:

```
{
   "name": "novos-voos-conector",
   "config": {
        "connector.class":
```

```
"io.debezium.connector.mongodb.MongoDbConnector",
        "mongodb.connection.string": "mongodb://mongodb:27017",
        "mongodb.user": "user",
        "mongodb.password": "password",
        "database.include.list": "livro-kafka",
        "collection.include.list": "livro-kafka.voos",
        "key.converter": "org.apache.kafka.connect.json.JsonConverter",
        "key.converter.schemas.enable": "false",
        "skipped.operations": "u,d,t",
        "topic.prefix": "ms-companhias-aereas"
   }
}
```

E O NOME DO TÓPICO?

Por padrão, os *connector plugins* concatenam algumas configurações presentes no conector para formarem o nome do tópico. No caso do *connector plugin* para MongoDB, considera-se as seguintes:

- `topic.prefix`
- `collection.include.list`

Da forma com que o conector foi configurado, a mensagem será postada, portanto, no tópico `ms-companhas-aereas.livro-kafka.voos`. Entretanto, se tivéssemos duas *collections* mapeadas na configuração `collection.include.list`, as mensagens iriam para dois tópicos diferentes.

Com o nome do tópico determinado, nosso diagrama de solução passa a ser assim:

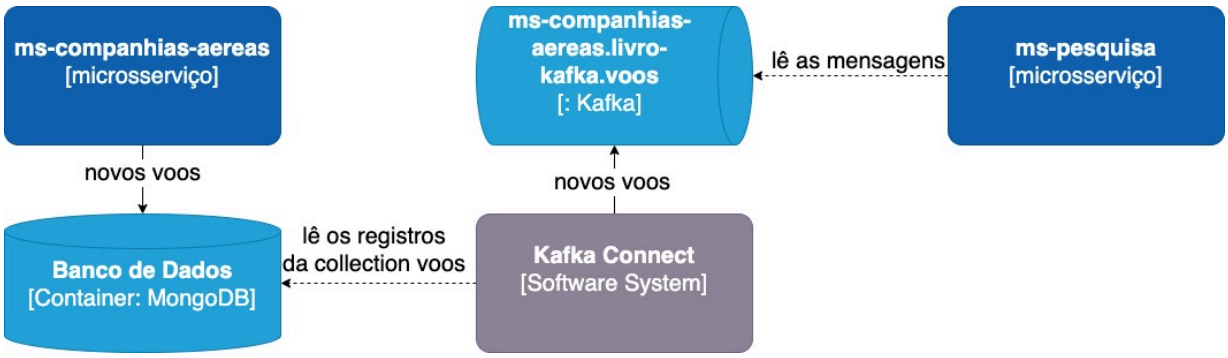


Figura 3.8: Diagrama atualizado com o nome correto do tópico.

Sobrescrita de configurações no conector

Como vimos, podemos subir o Kafka Connect com algumas configurações padrões para todos os conectores instalados. Entretanto, caso seja necessário, podemos sobrescrever essas configurações no conector. Um exemplo é a configuração `key.converter`. Ela foi configurada no Kafka Connect com o valor `org.apache.kafka.connect.storage.StringConverter`.

Com essa configuração, no caso do MongoDB, a chave será persistida no formato a seguir, onde `64ea327536818a32105b7ca8` é o valor da chave:

```
Struct{id={"$oid": "64ea327536818a32105b7ca8"}}
```

Ao sobrescrevermos essa configuração no conector com o valor `org.apache.kafka.connect.json.JsonConverter`, a chave será persistida no formato a seguir:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "id"
      }
```

```
    ],
    "optional": false,
    "name": "topico.livro-kafka.voos.Key"
  },
  "payload": {
    "id": {
      "$oid": "64ea327536818a32105b7ca8"
    }
  }
}
```

que contém não somente o valor da chave, mas também o *JSON schema* da chave. Essa pode ser uma informação importante para algumas aplicações. Mas também pode acarretar maior ocupação em disco das mensagens enviadas.

Para que não seja enviado o *JSON Schema* na mensagem, pode-se usar a configuração `key.converter.schemas.enable` como `false`. Com ela, a mensagem ficará no formato:

```
{"id":"{"$oid": "64ea307572c0aa19692e648a"}"}
```

que pode ser bem mais fácil de ser lido por *parsers* de JSON. De qualquer forma, antes de sobrescrever alguma configuração previamente estabelecida no Kafka Connect, verifique se essa não atende às necessidades da aplicação.

3.5 Subindo a infraestrutura da aplicação

A infraestrutura do Alura Viagens consiste dos arquivos a seguir:

- o `docker-compose.yml`;
- o script `infra-up.sh`, que sobe os contêineres docker e instala os conectores;
- o script `infra-down.sh`, que derruba os contêineres criados no script anterior e remove possíveis volumes usados.

Esses arquivos encontram-se na pasta `docker` na raiz do projeto.

Ao rodarmos o `infra-up.sh` veremos as seguintes mensagens no terminal:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % ./infra-up.sh
Subindo a infra da aplicação...
Creating network "bubble" with driver "bridge"
Creating zookeeper ... done
Creating mongodb   ... done
Creating kafka     ... done
Creating criador-topicos ... done
Creating kafka-connect   ... done
Creating redpanda        ... done
Removendo conteiner criador-topicos...
criador-topicos
Instalando conectores no kafka connect...
{"name":"novos-voos-conector","config":
{"connector.class":"io.debezium.connector.mongodb.MongoDbConnector","mongo
db.connection.string":"mongodb://mongodb:27017","mongodb.user":"user","mon
godb.password":"password","database.include.list":"livro-
kafka","collection.include.list":"livro-
kafka.voos","key.converter":"org.apache.kafka.connect.json.JsonConverter",
"key.converter.schemas.enable":"false","skipped.operations":"u,d,t","topic
.prefix":"topico","name":"novos-voos-conector"},"tasks":
[],"type":"source"}%
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker %
```

Repare que, assim que a infraestrutura entra no ar, o conector que criamos é instalado no Kafka Connect.

> Caso esse último passo não tenha acontecido, acesse a pasta `docker/cdc-kafka-connect` e rode o script `novos-voos-conector.sh`. Se mesmo assim o conector não for criado, vale a pena verificar as configurações de memória e processador do docker instalado na sua máquina.

Podemos verificar se o Kafka Connect está no ar acessando o endereço *http://localhost:8083*. A mensagem exibida será semelhante à seguinte:

```
{"version":"7.4.1-
ccs","commit":"fed9c006bfc7ba5bf7d2dee840e041d1a851d903","kafka_cluster_id
":"QtK-U92GQ2KxZ4RvBf2YVQ"}
```

Ao subirmos a infra da nossa aplicação, é possível notar que o *script* subiu o conector *novos-voos-connector*. Vamos verificar se isso realmente aconteceu, usando a Kafka Connect REST API.

3.6 A Kafka Connect REST API

O Kafka Connect possui uma API REST bem simples, que pode ser considerada, a princípio, como um CRUD de conectores, pois com ela é possível realizarmos operações de inserção, busca, atualização e remoção de conectores. Para acessarmos ou manipularmos os conectores, devemos usar o recurso `/connectors`. A operação será determinada de acordo com o verbo utilizado e os demais dados de entrada.

Veremos alguns exemplos usando o conector recém-criado e o comando cURL:

- `GET /connector` : retorna uma lista com os nomes de todos os conectores instalados.

Essa consulta nos retorna o seguinte resultado:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl -X GET
http://localhost:8083/connectors
["novos-voos-conector"]
```

- `GET /connector/<<nome-do-conector>>` : retorna informações sobre um conector específico.

Se quisermos, portanto, verificar os dados do nosso conector:

```
curl -X GET http://localhost:8083/connectors/novos-voos-conector
{"name":"novos-voos-conector","config":
{"connector.class":"io.debezium.connector.mongodb.MongoDbConnector","key.converter.schemas.enable":"false","collection.include.list":"livro-kafka.voos","topic.prefix":"topico","mongodb.connection.string":"mongodb://mongodb:27017","mongodb.password":"password","mongodb.user":"user","name":"novos-voos-conector","skipped.operations":"u,d,t","key.converter":"org.apache.kafka.connect.json.JsonConverter","database.include.list":"livro-kafka"},"tasks":[{"connector":"novos-voos-conector","task":0}],"type":"source"}%
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker %
```

- `POST /connector` : instala um conector no Kafka Connect

Esse foi o *endpoint* que usamos para instalar o conector com a base do *ms-companhias-aereas*. Ele foi usado no script `docker/cdc-kafka-connect/novos-voos-conector.sh` , mostrado a seguir:

```
curl -X POST \
  http://localhost:8083/connectors/ \
  -H 'content-type: application/json' \
  -d @novos-voos-conector.json
```

- `DELETE /connector/<<nome-do-conector>>` : remove o conector do Kafka Connect.

Há um exemplo de uso desse endpoint no *script* `docker/cdc-kafka-connect/novos-voos-conector-remover.sh` . Seu conteúdo está a seguir:

```
curl -X DELETE http://localhost:8083/connectors/novos-voos-conector
```

A relação completa de todos os *endpoints* dessa API pode ser acessada pelo seguinte endereço: https://docs.confluent.io/platform/current/connect/references/restapi.html.

3.7 Testando a solução

Com a infraestrutura no ar, vamos então:

- rodar o *ms-companhias-aereas*;
- usar sua API REST para cadastrar um novo voo;
- verificar banco de dados e o tópico `ms-companhias-aereas.livro.kafka.voos` . Uma vez que o novo voo estiver cadastrado, poderemos implementar o *ms-pesquisa*.

Para subir o *ms-companhias-aereas*, podemos usar nossa IDE predileta ou mesmo a linha de comando. Na raiz do projeto, digite:

```
mvn -f ms-companhias-aereas/pom.xml compile spring-boot:start
```

Com a aplicação no ar, podemos cadastrar um novo voo. Para isso, podemos usar o comando cURL a seguir:

```
curl --location 'http://localhost:8080/voos' \
--header 'Content-Type: application/json' \
--data '{
   "origem":"CGH",
   "partida":"2024-08-14T18:38:39.072390",
   "destino":"SDU",
   "chegada":"2024-08-14T20:38:39.072451",
   "companhiaAerea":"GOL",
   "precoAdulto": 1000.00,
   "precoCrianca": 500.00
}'
```

> Se preferir, há também um projeto do Postman presente na raiz do projeto que contém essa requisição. É o arquivo `livro-kafka.postman_collection.json` .

Para visualizarmos o banco de dados, usaremos o DBSchema, que será o nosso *client* de banco de dados para todos os bancos que

utilizarmos ao longo do livro. Ao buscarmos todos os voos cadastrados, podemos ver que o voo acima está no banco:

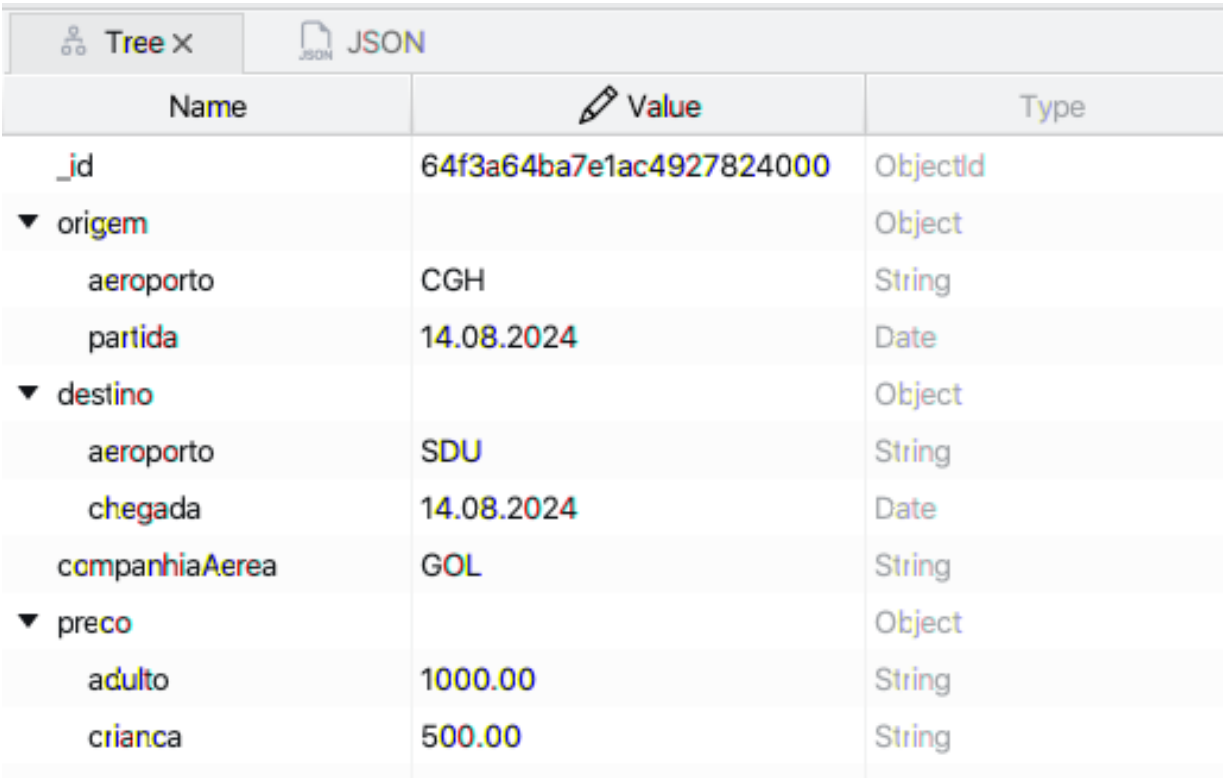


Figura 3.9: Voo cadastrado no MongoDB.

Por último, precisamos visualizar o tópico `ms-companhias-aereas.livro-kafka.voos` , para ver se a mensagem chegou até ele via Kafka Connect.

Atualmente há diversos *clients* para Kafka. Alguns podem ser instalados no seu desktop, como o **Conduktor** (https://www.conduktor.io/desktop/) ou o **Kafka Tool** (https://www.kafkatool.com). Outros são acessíveis via *browser*, e rodam via contêineres docker, como o **Kafdrop** (https://github.com/obsidiandynamics/kafdrop). Na nossa solução, usaremos o **Redpanda Console** (https://redpanda.com/redpanda-console-kafka-ui), que provê não somente um acesso mais detalhado aos tópicos e às mensagens contidas neles, mas também ao Kafka Connect e outros recursos que usaremos ao longo deste livro.

Para utilizarmos o Redpanda Console, precisamos inserir algumas configurações que façam com que ele se conecte ao Kafka. Para isso, escrevemos o arquivo `/tmp/config.yaml` no contêiner, contendo o endereço do Kafka, da seguinte forma:

```
command: -c "echo \"$$CONSOLE_CONFIG_FILE\" > /tmp/config.yml;
/app/console"
environment:
    CONFIG_FILEPATH: /tmp/config.yml
    CONSOLE_CONFIG_FILE: |
      kafka:
        brokers: ["kafka:29092"]
```

A configuração completa do Redpanda está a seguir, no `docker-compose.yml`:

```
redpanda:
  image: docker.redpanda.com/redpandadata/console:v2.3.0
  container_name: redpanda
  ports:
    - 18080:8080
  depends_on:
      - kafka
      - kafka-connect
  restart: always
```

```
  entrypoint: /bin/sh
  command: -c "echo \"$$CONSOLE_CONFIG_FILE\" > /tmp/config.yml;
/app/console"
  environment:
      CONFIG_FILEPATH: /tmp/config.yml
      CONSOLE_CONFIG_FILE: |
        kafka:
          brokers: ["kafka:29092"]
  networks:
      bubble:
```

Para acessarmos o Redpanda Console, usamos o endereço *http://localhost:18080*. E, na página principal, clicar no link *Topics*:

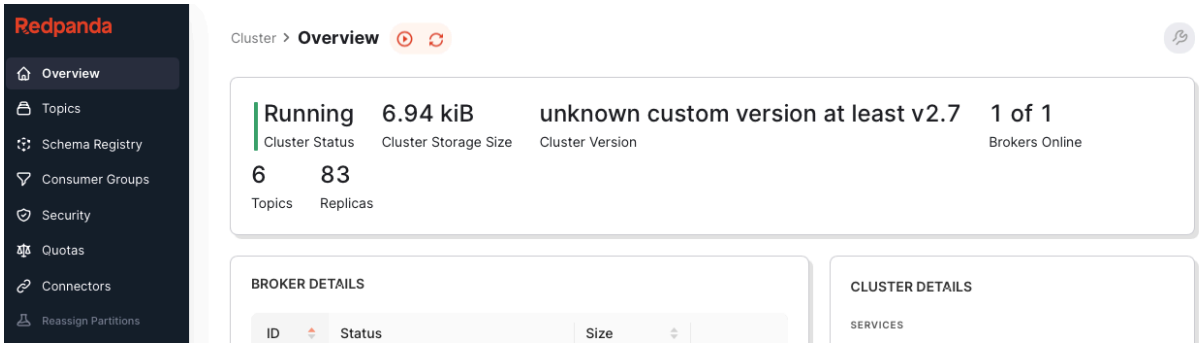


Figura 3.10: Trecho da página principal do Redpanda Console.

A lista de tópicos existentes no Kafka será exibida.

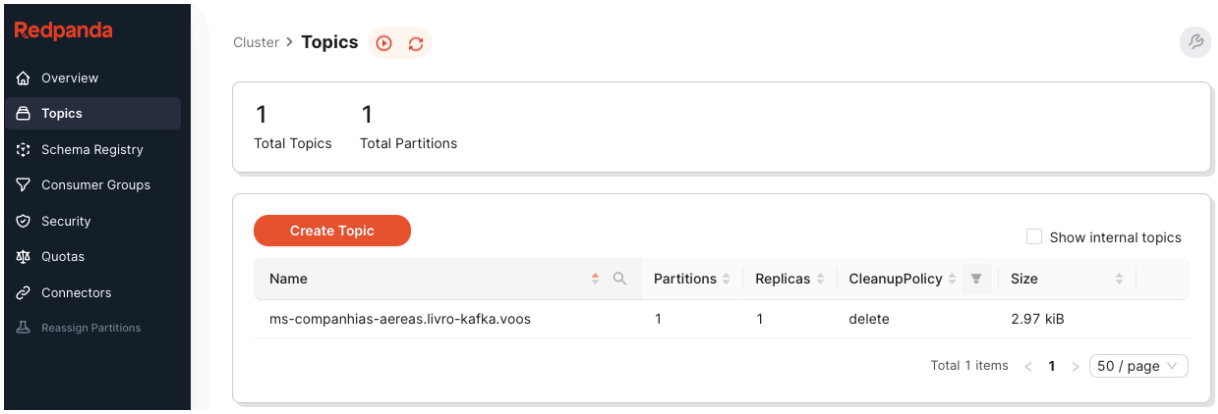


Figura 3.11: Trecho da página do Redpanda Console, destacando a lista de tópicos existentes.

Caso seja selecionado o *checkbox* **show internal topics** serão também exibidos os tópicos de controle do Kafka Connect.

Ao selecionar o tópico `ms-companhias-aereas.livro-kafka.voos`, veremos o novo voo cadastrado. É possível observar detalhes da mensagem, clicando no botão **+** ao lado esquerdo da mensagem. Ela contém diversas informações, como o *JSON Schema* da mensagem e informações sobre o evento capturado pelo Kafka Connect. Mas vamos nos concentrar no atributo *payload*, que contém as informações dos voos enviados:


```
"payload": {
    "before": null,
    "after": "{\"_id\": {\"$oid\":
\"64f3a64ba7e1ac4927824000\"},\"origem\": {\"aeroporto\":
\"CGH\",\"partida\": {\"$date\": 1723671519000}},\"destino\":
{\"aeroporto\": \"SDU\",\"chegada\": {\"$date\":
1723678719000}},\"companhiaAerea\": \"GOL\",\"preco\": {\"adulto\":
\"1000.00\",\"crianca\": \"500.00\"},\"_class\":
\"br.com.alura.mscompanhiasaereas.dominio.Voo\"}",
    "updateDescription": null,
    "source": {
        "version": "2.3.0.Final",
        "connector": "mongodb",
        "name": "ms-companhias-aereas",
        "ts_ms": 1693689419000,
        "snapshot": "false",
        "db": "livro-kafka",
        "sequence": null,
        "rs": "rs0",
        "collection": "voos",
        "ord": 1,
        "lsid": null,
        "txnNumber": null,
        "wallTime": null
    },
    "op": "c",
```

```
    "ts_ms": 1693689420029,
    "transaction": null
}
```

Repare que o elemento `after` do json da mensagem contém os valores presentes no banco de dados. Logo, o Kafka Connect e o conector que criamos cumpriram o seu papel. Há também diversas outras informações no elemento `payload` . Essas serão vistas no próximo capítulo.

3.8 Resumo

Neste capítulo vimos como implementar a integração do Kafka Connect com o banco de dados MongoDB. Também testamos essa integração, fazendo com que o *ms-pesquisa*, quando implementado, consiga receber os voos criados pelas companhias aéreas e ofertá-los aos clientes do Alura Viagens.

No próximo capítulo, veremos como o Kafka Connect possui mecanismos que buscam garantir a resiliência da solução como um todo. Também veremos como ele se integra ao MongoDB e implementaremos mais uma funcionalidade no *ms-companhias-aereas* antes de partirmos para a implementação do *ms-pesquisa*.

Kafka Connect e seus recursos

Até o momento, vimos algumas definições básicas sobre o Kafka Connect. Vimos como instalar os *connector plugins* e também testamos o seu uso numa integração entre uma *collection* do MongoDB e o Kafka, enviando mensagens para um tópico.

A seguir, veremos algumas das principais características do Kafka Connect que nos ajudam a tornar nossas soluções mais resilientes. Também criaremos integrações entre outras fontes de dados e usaremos os recursos de *Single Message Transformations (SMTs)*, que nos fornecem diversas funcionalidades. Essas características nos serão muito úteis na comunicação entre os microsserviços existentes e nos que veremos mais à frente.

Esperamos que sua viagem esteja sendo prazerosa!

Capítulo 4
Kafka Connect e a resiliência das aplicações

O *ms-companhias-aereas* utiliza o banco de dados MongoDB como mecanismo de persistência. Logo, para que o Kafka Connect funcione, ele precisará se conectar a esse banco, mais precisamente na *collection* **voos**. Uma vez que sejam persistidos novos voos nessa *collection* ele poderá enviá-los para o Kafka. Temos então a seguinte arquitetura:

Figura 4.1: Arquitetura da solução até o momento.

Além disso, como já vimos no capítulo anterior, em arquiteturas que envolvam mais de uma aplicação, precisamos nos preocupar com cenários onde uma ou mais aplicações falharem. Dessa forma, fazemos com que nossa arquitetura seja mais resiliente.

> Segundo o Dicionário Aurélio (2023), resiliência refere-se tanto à elasticidade que faz com que certos corpos deformados voltem à sua forma original quanto à capacidade de rápida adaptação ou recuperação.
>
> Em Engenharia de Software, um dos livros mais importantes a abordar esse tema foi o *Release It! Design and Deploy Production-Ready Software* (2007) de Michael T. Nygard. Nele, o autor discute a importância de projetar sistemas que sejam capazes de se recuperar de falhas e erros de maneira eficaz.
>
> É possível também encontrarmos preocupações relacionadas à resiliência de aplicações em livros como *Building Microservices* (2021) e *Microservices Patterns* (2019).

Nessa situação, poderíamos ter os seguintes cenários:

Cenário 1: Banco de dados fora do ar ou inacessível pelo *ms-companhias-aereas*

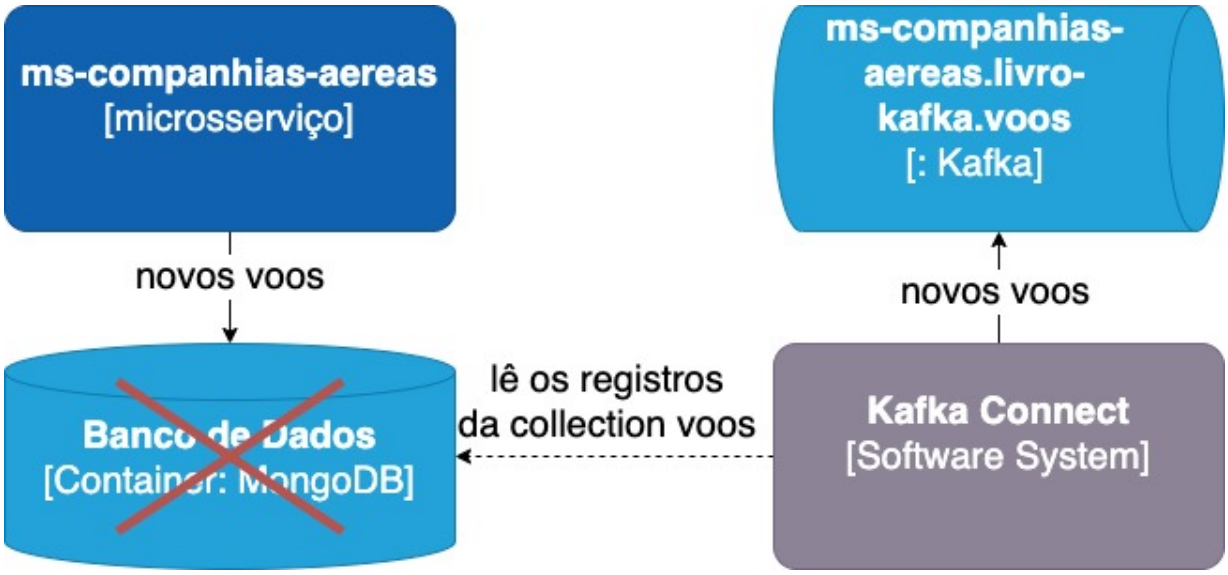


Figura 4.2: MongoDB com problemas.

Nesse caso, o *ms-companhias-aereas* não criará nenhum voo novo. O Kafka Connect não escutará nenhum evento, e os desenvolvedores do *ms-companhias-aereas* precisam entender o que deve ser feito nessa situação. Eles podem simplesmente fazer com que o microsserviço lance um erro aos seus clientes informando que o sistema está fora do ar. Podem também preparar o banco de dados para que possua algum tipo de redundância.

Cenário 2: *ms-companhias-aereas* fora do ar

Figura 4.3: ms-companhias-aereas com problemas.

Com o microsserviço fora do ar, não haverá novos voos. Assim como o banco de dados, é possível que políticas de replicação ajudem a manter pelo menos 1 instância no ar, caso as outras não estejam. Pode-se verificar também possíveis *bugs* relacionados a estouro de memória ou *loops* infinitos ou desnecessários.

Cenário 3: Kafka fora do ar

Figura 4.4: Kafka com problemas.

No caso de o Kafka estar fora do ar, é necessário verificarmos questões relacionadas ao Kafka Connect. Ele não terá como enviar as mensagens para o Kafka, pois ele não os receberá.

Como o Kafka Connect se comportará nesse cenário? Para verificarmos isso, podemos subir a nossa solução e, assim que tudo estiver no ar, derrubar deliberadamente o Kafka.

Vamos então rodar o *infra-up.sh*:

```
Subindo a infra da aplicação...
Creating network "bubble" with driver "bridge"
Creating mongodb   ... done
Creating zookeeper ... done
Creating kafka     ... done
Creating kafka-connect   ... done
Creating criador-topicos ... done
Creating redpanda        ... done
Removendo conteiner criador-topicos...
criador-topicos
Instalando conectores no kafka connect...
{"name":"novos-voos-conector","config":
{"connector.class":"io.debezium.connector.mongodb.MongoDbConnector","mongo
```

db.connection.string":"mongodb://mongodb:27017","mongodb.user":"user","mongodb.password":"password","database.include.list":"livro-kafka","collection.include.list":"livro-kafka.voos","key.converter":"org.apache.kafka.connect.json.JsonConverter","key.converter.schemas.enable":"false","skipped.operations":"u,d,t","topic.prefix":"ms-companhias-aereas","name":"novos-voos-conector"},"tasks":[],"type":"source"}%
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker %

Vamos também rodar o *ms-companhias-aereas*. Podemos fazer isso tanto via nossa IDE favorita quanto via linha de comando. Vamos usar a segunda opção. Na pasta onde se encontra o projeto, vamos rodar o comando a seguir:

```
mvn spring-boot:start
```

Ao enviarmos uma mensagem usando o seguinte comando:

```
curl --location 'http://localhost:8080/voos' \
--header 'Content-Type: application/json' \
--data '{
   "origem":"CGH",
   "partida":"2024-08-14T18:38:39.072390",
   "destino":"SDU",
   "chegada":"2024-08-14T20:38:39.072451",
   "companhiaAerea":"GOL",
   "precoAdulto": 1000.00,
   "precoCrianca": 500.00
}'
```

podemos ver no Redpanda a mensagem chegando ao Kafka sem problemas.


```
"payload": {
    "before": null,
    "after": "{\"_id\": {\"$oid\":
\"64fb4ca9e22d0331a5716cc7\"},\"origem\": {\"aeroporto\":
\"CGH\",\"partida\": {\"$date\": 1723671519000}},\"destino\":
{\"aeroporto\": \"SDU\",\"chegada\": {\"$date\":
1723678719000}},\"companhiaAerea\": \"GOL\",\"preco\": {\"adulto\":
```

```
\"1000.00\",\"crianca\": \"500.00\"},\"_class\":
\"br.com.alura.mscompanhiasaereas.dominio.Voo\"}",
    "updateDescription": null,
    "source": {
        "version": "2.3.0.Final",
        "connector": "mongodb",
        "name": "ms-companhias-aereas",
        "ts_ms": 1694190761000,
        "snapshot": "false",
        "db": "livro-kafka",
        "sequence": null,
        "rs": "rs0",
        "collection": "voos",
        "ord": 1,
        "lsid": null,
        "txnNumber": null,
        "wallTime": null
    },
    "op": "c",
    "ts_ms": 1694190761779,
    "transaction": null
}
```


Assim como fizemos no capítulo anterior, aqui vemos apenas um trecho da mensagem, por brevidade.

Ao rodarmos o comando `docker ps`, vemos algo parecido com a resposta a seguir:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED          STATUS                     PORTS
NAMES
fe20a292ca49   docker.redpanda.com/redpandadata/console:v2.3.0   "/bin/sh
-c 'echo \"$…"   10 minutes ago   Up 9 minutes              0.0.0.0:18080-
>8080/tcp                                redpanda
125ed1e1e919   cdc-kafka-connect                                  "bash -c
'echo \">>>>…"   10 minutes ago   Up 10 minutes (healthy)   0.0.0.0:8083-
```

```
>8083/tcp, 9092/tcp                         kafka-connect
0ce4ce77e5fc   confluentinc/cp-server:7.3.3
"/etc/confluent/dock…"    10 minutes ago   Up 10 minutes
0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->9092/tcp        kafka
379eef94ac18   mongo:4.2                                          "docker-
entrypoint.s…"    10 minutes ago   Up 10 minutes (healthy)
0.0.0.0:27017->27017/tcp                                mongodb
81c01ade6e64   zookeeper:3.5                                      "/docker-
entrypoint.…"    10 minutes ago   Up 10 minutes             2888/tcp,
3888/tcp, 0.0.0.0:2181->2181/tcp, 8080/tcp   zookeeper
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker %
```

É possível notar que o contêiner correspondente ao Kafka é o `0ce4ce77e5fc`. Vamos pará-lo:

```
docker stop 0ce4ce77e5fc
```

Naturalmente, com o Kafka fora do ar, as mensagens não chegarão ao tópico. Além disso, é possível visualizar no *log* do Kafka Connect que ele perdeu conexão com o Kafka:

```
WARN [Consumer clientId=kafka-connect--configs, groupId=kafka-connect]
Connection to node 1001 (kafka/172.21.0.4:29092) could not be established.
Broker may not be available. (org.apache.kafka.clients.NetworkClient)
```

> Para visualizar o *log* do Kafka Connect, existem duas formas:
>
> - via Docker Desktop, acessando o container específico;
> - via linha de comando, com `docker logs -f <<id do container>>`. No nosso exemplo, o id do contêiner da ferramenta é o `125ed1e1e919`.

Assim que recebermos a confirmação do Docker de que o contêiner parou de fato, vamos enviar mais 4 mensagens via *curl*. E em seguida, subir o contêiner do Kafka novamente:

```
docker start 0ce4ce77e5fc
```

Assim que o Kafka voltar ao ar, o Kafka Connect fará verificações nos bancos de dados a que ele está conectado, em busca de mudanças que não foram enviadas para o Kafka, enviando as mensagens restantes logo em seguida, o que totalizará em 5 mensagens:

	Offset	Partition	Timestamp	Key	Value Preview	
+	4	0	08/09/2023, 14:10:30	{"id":"{\"$oid\": \"64fb55583e4dad078c035...	{"schema":{"ty…	
+	3	0	08/09/2023, 14:10:30	{"id":"{\"$oid\": \"64fb55573e4dad078c035...	{"schema":{"ty…	
+	2	0	08/09/2023, 14:10:30	{"id":"{\"$oid\": \"64fb55563e4dad078c035...	{"schema":{"ty…	
+	1	0	08/09/2023, 14:10:30	{"id":"{\"$oid\": \"64fb55553e4dad078c035...	{"schema":{"ty…	
+	0	0	08/09/2023, 14:08:16	{"id":"{\"$oid\": \"64fb54ff3e4dad078c035...	{"schema":{"ty…	

Figura 4.5: Mensagens restantes chegando ao Kafka após ele ser restabelecido.

Repare que as 4 últimas mensagens possuem o mesmo *timestamp*, visto que foram enviadas ao mesmo tempo. Dessa forma, podemos comprovar que o Kafka Connect envia as mensagens ao Kafka assim que tudo volta ao normal.

Cenário 4: Kafka Connect fora do ar

Figura 4.6: Kafka Connect com problemas.

Com o Kafka Connect fora do ar, perdemos provavelmente nossa principal integração entre as aplicações. As mensagens continuarão a ser persistidas no banco, mas deixarão de ser enviadas ao Kafka.

Para testar o comportamento do Kafka Connect, vamos seguir procedimentos semelhantes aos usados no cenário anterior:

- enviar um voo ao *ms-companhias-aereas*;
- interromper o contêiner do Kafka Connect;
- persistir mais 4 voos;
- subir o contêiner do Kafka Connect;
- verificar se o tópico `ms-companhias-aereas.livro-kafka.voos` contém as 5 mensagens.

O primeiro voo foi enviado ao tópico. Tudo certo.

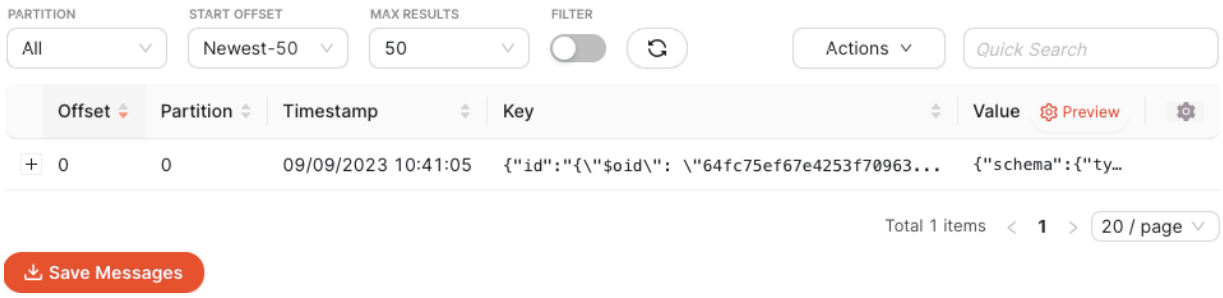


Figura 4.7: Tudo no ar. Voo enviado.

Vamos então verificar qual o id do contêiner do Kafka Connect e o derrubar:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED          STATUS                    PORTS
NAMES
acba6f7ba3b1   docker.redpanda.com/redpandadata/console:v2.3.0   "/bin/sh
-c 'echo \"$…"   10 minutes ago   Up 9 minutes              0.0.0.0:18080-
>8080/tcp                                redpanda
279f5816e6bd   cdc-kafka-connect                                  "bash -c
'echo \">>>>…"   10 minutes ago   Up 10 minutes (healthy)   0.0.0.0:8083-
>8083/tcp, 9092/tcp                      kafka-connect
```

```
d1f2c4f062bb   confluentinc/cp-server:7.3.3
"/etc/confluent/dock…"    10 minutes ago   Up 10 minutes
0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->9092/tcp        kafka
86a9c937ea37   mongo:4.2                                              "docker-
entrypoint.s…"    10 minutes ago   Up 10 minutes (healthy)
0.0.0.0:27017->27017/tcp                                mongodb
e99c915691c8   zookeeper:3.5                                          "/docker-
entrypoint.…"    10 minutes ago   Up 10 minutes             2888/tcp,
3888/tcp, 0.0.0.0:2181->2181/tcp, 8080/tcp   zookeeper
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % docker stop
279f5816e6bd
```

Após enviarmos mais 4 voos, subimos novamente o Kafka Connect:

```
docker start 279f5816e6bd
```

Assim que o Kafka Connect voltar ao ar, os registros do banco são lidos retroativamente. E, assim como no cenário anterior, o *timestamp* dos registros lidos após sua volta é praticamente o mesmo.

Offset	Partition	Timestamp	Key	Value Preview
4	0	09/09/2023 10:47:17	{"id":"{\"$oid\": \"64fc76ec67e4253f70963...	{"schema":{"ty…
3	0	09/09/2023 10:47:17	{"id":"{\"$oid\": \"64fc76eb67e4253f70963...	{"schema":{"ty…
2	0	09/09/2023 10:47:17	{"id":"{\"$oid\": \"64fc76ea67e4253f70963...	{"schema":{"ty…
1	0	09/09/2023 10:47:17	{"id":"{\"$oid\": \"64fc76e867e4253f70963...	{"schema":{"ty…
0	0	09/09/2023 10:41:05	{"id":"{\"$oid\": \"64fc75ef67e4253f70963...	{"schema":{"ty…

Figura 4.8: Tudo no ar novamente. Demais voos enviados ao tópico.

4.1 Leitura do banco de dados pelo Kafka Connect

Mas como o Kafka Connect lê os registros? Será que ele realiza consultas simples nas tabelas (ou *collections*) configuradas nos

conectores?

Se ele trabalhasse dessa forma, teríamos alguns problemas.

- se ocorrerem eventos de inserção, atualização e remoção de registros em um banco com o Kafka Connect fora do ar, eles nunca seriam lidos, visto que eles existiriam sem que o Kafka Connect pudesse lê-los.
- podemos inclusive excluir a tabela do banco de dados, e os registros presentes nela deixariam de ser enviados para o Kafka.
- ao fazer com que mais de uma aplicação realize buscas nas tabelas do banco de dados, podemos fazer com que ele perca performance.

Então, onde ele realiza as suas leituras?

CAPTURA DE MUDANÇAS EM TEMPO REAL DOS DADOS DO BANCO COM CONECTORES CDC (*CHANGE DATA CAPTURE*)

Conforme vimos, podemos ter problemas que fazem com que a ideia de fazer o conector ler os dados diretamente das tabelas do banco não seja muito boa.

Entretanto, os bancos de dados possuem também **logs** de todas as tarefas executadas por eles. E, aconteça o que acontecer com as tabelas e seus dados, os logs do banco permanecem intactos, fazendo com que tenhamos uma linha do tempo descrevendo todas as operações executadas por ele. Mesmo com dados que não existem mais.

Os conectores CDC, portanto, têm como base a leitura desses logs, e cada banco de dados fornece uma maneira de lê-los de forma segura.

Existem também os conectores *JDBC Source*. Esses, ao contrário dos conectores que estamos usando na nossa solução, leem os dados diretamente das tabelas do banco de dados, como o próprio nome diz, acessam bancos de dados relacionais usando JDBC e podem ser uma boa alternativa para o caso de migrações simples de dados. Entretanto, se o objetivo da solução é capturar eventos, devemos usar os conectores CDC.

Leitura de dados do MongoDB pelos conectores CDC

O MongoDB oferece diversas restrições à leitura de seus logs. Dessa forma, o log do banco fica protegido de alterações ou mesmo exclusões acidentais. Em contrapartida, oferece o mecanismo de *change streams*.

O *change streams* é uma funcionalidade poderosa do MongoDB que permite às aplicações monitorarem e reagirem a mudanças nos dados de uma *collection* em tempo real. Assim, o Kafka Connect consegue capturar esses eventos e os enviar ao Kafka.

Podemos simular o comportamento do Kafka Connect acessando o MongoDB, da seguinte maneira:

- acessar o contêiner do MongoDB:

```
docker exec -it mongodb bash
```

- abrir o *client* de MongoDB por linha de comando:

```
mongo --username user --password password --authenticationDatabase livro-kafka --host localhost:27017
```

- e, logo em seguida, escolher o banco de dados da aplicação:

```
use livro-kafka
```

- por último, precisamos acessar a *collection* de voos, usando os recursos de *change streams* do MongoDB. Para isso, usamos o método `watch()` ouvindo todas as manipulações que ocorrerem no banco por 20 segundos:

```
db.getCollection('voos').watch([], {maxAwaitTimeMS: 20000})
```

Em seguida, podemos realizar operações de cadastro e exclusão de voos usando o *ms-companhias-aereas*. Mas podemos também rodar os testes unitários descritos na classe `RepositorioDeVoosTest`. Ele possui testes que envolvem operações de inclusão e remoção de voos no banco.

```
@SpringBootTest
class RepositorioDeVoosTest {

    @Autowired
    private RepositorioDeVoos repositorioDeVoos;
```

```
    @Autowired
    private MongoTemplate mongoTemplate;

    private Voo voo;

    @BeforeEach
    void setUp() {
        voo = doRioParaBeloHorizonteAmanha();
        repositorioDeVoos.save(voo);
    }

    @AfterEach
    void tearDown(){
        mongoTemplate.remove(voo);
    }

    @Test
    void cadastrarUmVoo() {
        mongoTemplate.findAll(Voo.class);

        assertThat(mongoTemplate.findAll(Voo.class)).contains(voo);
    }

    @Test
    void removerVoo(){
        repositorioDeVoos.deleteById(voo.getId());

        assertThat(mongoTemplate.findAll(Voo.class)).doesNotContain(voo);
    }
}
```

Ao rodarmos os testes, veremos no console resultados semelhantes aos seguintes:

```
/* 1 */
{
    "_id" : {
        "_data" :
"8265063289000000012B022C0100296E5A1004EFBB989D38B64E14A2D5AF444E085284466
45F696400646506328610E667EF139845E0004"
```

```
    },
    "operationType" : "insert",
    "clusterTime" : Timestamp(1694904969, 1),
    "fullDocument" : {
        "_id" : ObjectId("65063288610e667ef139845e"),
        "origem" : {
            "aeroporto" : "SDU",
            "partida" : ISODate("2023-09-17T22:56:08.000Z")
        },
        "destino" : {
            "aeroporto" : "BSB",
            "chegada" : ISODate("2023-09-18T01:56:08.000Z")
        },
        "companhiaAerea" : "AZUL",
        "preco" : {
            "adulto" : "1000.00",
            "crianca" : "500.00"
        },
        "_class" : "br.com.alura.mscompanhiasaereas.dominio.Voo"
    },
    "ns" : {
        "db" : "livro-kafka",
        "coll" : "voos"
    },
    "documentKey" : {
        "_id" : ObjectId("65063288610e667ef139845e")
    }
}
/* 2 */
{
    "_id" : {
        "_data" :
"826506328A000000012B022C0100296E5A1004EFBB989D38B64E14A2D5AF444E085284466
45F696400646506328610E667EF139845E0004"
    },
    "operationType" : "delete",
    "clusterTime" : Timestamp(1694904970, 1),
    "ns" : {
        "db" : "livro-kafka",
```

```
        "coll" : "voos"
    },
    "documentKey" : {
        "_id" : ObjectId("65063288610e667ef139845e")
    }
}
/* 3 */
{
    "_id" : {
        "_data" :
"826506328A000000022B022C0100296E5A1004EFBB989D38B64E14A2D5AF444E085284466
45F696400646506328A610E667EF139845F0004"
    },
    "operationType" : "insert",
    "clusterTime" : Timestamp(1694904970, 2),
    "fullDocument" : {
        "_id" : ObjectId("6506328a610e667ef139845f"),
        "origem" : {
            "aeroporto" : "SDU",
            "partida" : ISODate("2023-09-17T22:56:10.000Z")
        },
        "destino" : {
            "aeroporto" : "BSB",
            "chegada" : ISODate("2023-09-18T01:56:10.000Z")
        },
        "companhiaAerea" : "AZUL",
        "preco" : {
            "adulto" : "1000.00",
            "crianca" : "500.00"
        },
        "_class" : "br.com.alura.mscompanhiasaereas.dominio.Voo"
    },
    "ns" : {
        "db" : "livro-kafka",
        "coll" : "voos"
    },
    "documentKey" : {
        "_id" : ObjectId("6506328a610e667ef139845f")
    }
```

```
}
/* 4 */
{
    "_id" : {
        "_data" :
"826506328A000000032B022C0100296E5A1004EFBB989D38B64E14A2D5AF444E085284466
45F696400646506328A610E667EF139845F0004"
    },
    "operationType" : "delete",
    "clusterTime" : Timestamp(1694904970, 3),
    "ns" : {
        "db" : "livro-kafka",
        "coll" : "voos"
    },
    "documentKey" : {
        "_id" : ObjectId("6506328a610e667ef139845f")
    }
}
```

Note que é possível ver as operações realizadas (inserção, remoção) bem como os dados envolvidos.

> Mais informações sobre o *change streams* podem ser obtidas em https://www.mongodb.com/docs/manual/changeStreams/
>
> Ao longo deste livro, será explicado como o Kafka Connect se conecta a todos os bancos de dados usados na nossa solução.

4.2 Um cenário mais específico - cancelamento de novos voos com os Tombstones Events

Na nossa solução, o Kafka Connect ouve apenas os eventos de inserção de voos no MongoDB. Mas as companhias aéreas, devido a problemas como a falta de compra de passagens para determinados

destinos, a falta de aeronaves ou mesmo de tripulação, precisam também cancelar voos. E para o Alura Viagens, seria um grande problema ofertar voos que não são mais suportados pelas companhias.

É necessário, portanto, que nossa solução implemente a funcionalidade de cancelamento de voos. Com ela, caso algum voo seja cancelado, ele será também removido dos sistemas da Alura Viagens. Para isso, precisamos realizar as seguintes alterações na solução:

- alterar a API de cadastro de novos voos (no *ms-companhias-aereas*) para não somente retornar o *http status code*, mas também uma identificação do voo cadastrado. Para facilitar o desenvolvimento, vamos usar o id gerado pelo banco de dados como identificador do voo;
- também no *ms-companhias-aereas* criar uma API de remoção de novos voos;
- alterar o conector instalado no Kafka Connect para que ele escute também operações de remoção de voos do banco, não somente de inserção.

Essas alterações estão presentes na *branch cap-4*. Então, para que elas sejam refletidas na aplicação, precisamos apenas executar o comando:

```
git checkout cap-4
```

Além das alterações nas APIs do *ms-companhias-aereas*, a alteração no conector é bem simples. Basta vermos o arquivo `docker/cdc-kafka-connect/novos-voos-conector.json` . Na linha onde se encontrava a configuração `"skipped.operations": "u,d,t",` , ela passa a ser `"skipped.operations": "u,t",` .

Removemos a operação de remoção de registros do banco como operação a ser ignorada pelo conector. O teste das novas funcionalidades deve, portanto, seguir os seguintes passos:

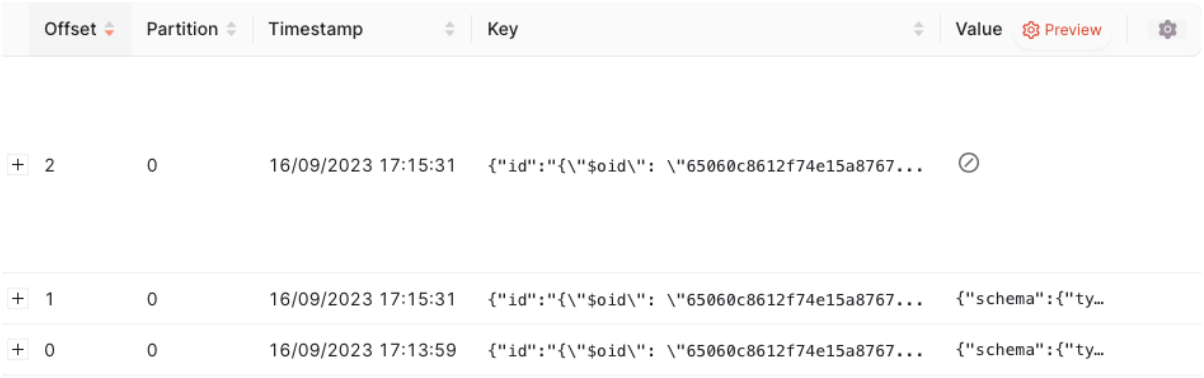
- cadastrar um novo voo e verificar se uma mensagem de persistência desse voo no banco foi enviada ao tópico;
- cancelar o voo que acabamos de cadastrar e verificar se uma mensagem de exclusão do voo do banco foi enviada ao tópico.

Ao realizarmos esses passos, notamos que novos registros foram gerados no tópico `ms-companhias-aereas.livro-kafka.voos`. Um deles está relacionado à remoção do registro no banco. Note a operação `"op":"d"` presente na mensagem de exclusão:


```
"payload": {
    "before": null,
    "after": null,
    "updateDescription": null,
    "source": {
        "version": "2.3.0.Final",
        "connector": "mongodb",
        "name": "ms-companhias-aereas",
        "ts_ms": 1694895330000,
        "snapshot": "false",
        "db": "livro-kafka",
        "sequence": null,
        "rs": "rs0",
        "collection": "voos",
        "ord": 1,
        "lsid": null,
        "txnNumber": null,
        "wallTime": null
    },
    "op": "d",
    "ts_ms": 1694895330956,
    "transaction": null
}
```

Podemos inclusive fazer mais um teste de resiliência da solução, derrubando o Kafka Connect, inserindo e removendo novos voos, e só depois subir novamente a aplicação. Se tudo der certo, os eventos de inserção e exclusão de voos estarão no tópico Kafka assim que o Kafka Connect voltar ao ar.

Mas é possível notar que mais um evento foi enviado ao tópico, além do evento anterior.



Offset	Partition	Timestamp	Key	Value
2	0	16/09/2023 17:15:31	{"id":"{\"$oid\": \"65060c8612f74e15a8767...	
1	0	16/09/2023 17:15:31	{"id":"{\"$oid\": \"65060c8612f74e15a8767...	{"schema":{"ty…
0	0	16/09/2023 17:13:59	{"id":"{\"$oid\": \"65060c8612f74e15a8767...	{"schema":{"ty…

Figura 4.9: Três eventos em vez de dois.

Podemos notar também esse evento não possui valor. Apenas a chave. Por quê?

Kafka, *tombstones events* e compactação de logs

O Kafka armazena as mensagens em disco como se fosse um log sequencial de informações, onde podemos entender que cada linha do log corresponde a uma mensagem. É possível, entretanto, configurarmos um tempo para que essas mensagens permaneçam disponíveis a todos os consumidores. E, ao final desse tempo, o Kafka deve tomar uma decisão quanto a elas:

- o Kafka pode simplesmente excluir as mensagens do tópico;
- o Kafka pode compactar essas mensagens.

Esse conceito é chamado de política de limpeza do tópico (*cleanup policy*).

Se optarmos pela compactação das mensagens do tópico, o Kafka buscará todas as mensagens que possuam chaves iguais e manterá no tópico apenas a mais recente. Isso fará com que o tamanho em disco ocupado seja menor. E, se houver mensagens com chaves exclusivas, essas serão mantidas. Entretanto, se a mensagem mais recente de um conjunto de mensagens com mesma chave não possuir valor, todas as mensagens serão removidas do tópico, inclusive a mais recente.

O Kafka Connect aproveita-se desse comportamento para manter o tópico o mais atualizado possível em relação ao banco de dados. Assim que ele recebe um evento do banco de remoção de registro, ele envia as duas mensagens citadas e, caso a política de limpeza relacionada ao tópico seja de compactação de mensagens, todas as mensagens com mesma chave e a mais recente com valor vazio serão removidas.

O Kafka Connect chama essa mensagem de *tombstone event*. É possível desabilitá-lo no conector, bastando configurar o elemento `tombstones.on.delete` para *false*. Entretanto, é recomendado não alterar o valor *default*, que é *true*. Assim, evita-se o armazenamento de mensagens que não representam o estado atual da informação no tópico.

> Mais sobre compactação de logs no Kafka pode ser encontrado no seguinte endereço:
> https://kafka.apache.org/documentation/#compaction

4.3 Resumo

Neste capítulo, pudemos ver como o Kafka Connect possui mecanismos que nos fornecem garantias de processamento das aplicações. Vimos também como ele realiza a leitura dos registros incluídos e excluídos do MongoDB. Implementamos o cancelamento de voos por parte do *ms-companhias-aereas*. Assim, pudemos ver como o Kafka Connect lida com a exclusão física de registros do banco de dados.

No próximo capítulo, começaremos a ver a implementação do *ms-pesquisa*. Continuaremos com nossa preocupação com a resiliência das aplicações. Então, buscaremos tratar possíveis problemas envolvendo o consumo de mensagens por esse microsserviço. Veremos também que, assim como na maneira como as **chaves** das mensagens são persistidas, é possível modificar os **valores** das mensagens, tornando-as mais "enxutas".

Capítulo 5
Melhorias nas mensagens postadas e o ms-pesquisa

No capítulo anterior, vimos que:

- quando criamos um novo voo, é inserido um registro no tópico `ms-companhias-aereas.livro-kafka.voos`;
- quando removemos um voo, são inseridos 2 registros. Um corresponde à operação de remoção do voo, e o outro, ao *tombstone event*.

Essas mensagens possuem muitas informações. Entre elas:

- a operação que se realizou no banco de dados que gerou a mensagem;
- informações do banco de dados, como versão, tipo e nome;
- tipo, obrigatoriedade e outras informações relacionadas a cada atributo do registro;
- ... e o registro em si.

A seguir, um exemplo completo de registro correspondente à inserção de um novo voo (obtido via *download* de mensagem do Redpanda):


```
{
    "partitionID": 0,
    "offset": 18,
    "timestamp": 1695385534584,
    "compression": "uncompressed",
    "isTransactional": false,
    "headers": [],
    "key": {
        "payload": {
            "id": "{\"$oid\": \"650d87bee0c6ff4b1a16a2a9\"}"
        },
```

```
    "encoding": "json",
    "schemaId": 0
},
"value": {
    "payload": {
        "schema": {
            "type": "struct",
            "fields": [
                {
                    "type": "string",
                    "optional": true,
                    "name": "io.debezium.data.Json",
                    "version": 1,
                    "field": "before"
                },
                {
                    "type": "string",
                    "optional": true,
                    "name": "io.debezium.data.Json",
                    "version": 1,
                    "field": "after"
                },
                {
                    "type": "struct",
                    "fields": [
                        {
                            "type": "array",
                            "items": {
                                "type": "string",
                                "optional": false
                            },
                            "optional": true,
                            "field": "removedFields"
                        },
                        {
                            "type": "string",
                            "optional": true,
                            "name": "io.debezium.data.Json",
                            "version": 1,
```

```
                            "field": "updatedFields"
                        },
                        {
                            "type": "array",
                            "items": {
                                "type": "struct",
                                "fields": [
                                    {
                                        "type": "string",
                                        "optional": false,
                                        "field": "field"
                                    },
                                    {
                                        "type": "int32",
                                        "optional": false,
                                        "field": "size"
                                    }
                                ],
                                "optional": false,
                                "name":
"io.debezium.connector.mongodb.changestream.truncatedarray",
                                "version": 1
                            },
                            "optional": true,
                            "field": "truncatedArrays"
                        }
                    ],
                    "optional": true,
                    "name":
"io.debezium.connector.mongodb.changestream.updatedescription",
                    "version": 1,
                    "field": "updateDescription"
                },
                {
                    "type": "struct",
                    "fields": [
                        {
                            "type": "string",
                            "optional": false,
```

```
                "field": "version"
            },
            {
                "type": "string",
                "optional": false,
                "field": "connector"
            },
            {
                "type": "string",
                "optional": false,
                "field": "name"
            },
            {
                "type": "int64",
                "optional": false,
                "field": "ts_ms"
            },
            {
                "type": "string",
                "optional": true,
                "name": "io.debezium.data.Enum",
                "version": 1,
                "parameters": {
                    "allowed":
"true,last,false,incremental"
                },
                "default": "false",
                "field": "snapshot"
            },
            {
                "type": "string",
                "optional": false,
                "field": "db"
            },
            {
                "type": "string",
                "optional": true,
                "field": "sequence"
            },
```

```
                {
                    "type": "string",
                    "optional": false,
                    "field": "rs"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "collection"
                },
                {
                    "type": "int32",
                    "optional": false,
                    "field": "ord"
                },
                {
                    "type": "string",
                    "optional": true,
                    "field": "lsid"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "txnNumber"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "wallTime"
                }
            ],
            "optional": false,
            "name": "io.debezium.connector.mongo.Source",
            "field": "source"
        },
        {
            "type": "string",
            "optional": true,
            "field": "op"
```

```
                    },
                    {
                        "type": "int64",
                        "optional": true,
                        "field": "ts_ms"
                    },
                    {
                        "type": "struct",
                        "fields": [
                            {
                                "type": "string",
                                "optional": false,
                                "field": "id"
                            },
                            {
                                "type": "int64",
                                "optional": false,
                                "field": "total_order"
                            },
                            {
                                "type": "int64",
                                "optional": false,
                                "field": "data_collection_order"
                            }
                        ],
                        "optional": true,
                        "name": "event.block",
                        "version": 1,
                        "field": "transaction"
                    }
                ],
                "optional": false,
                "name": "ms-companhias-aereas.livro-kafka.voos.Envelope"
            },
            "payload": {
                "before": null,
                "after": "{\"_id\": {\"$oid\":
\"650d87bee0c6ff4b1a16a2a9\"},\"origem\": {\"aeroporto\":
\"SDU\",\"partida\": {\"$date\": 1695471934000}},\"destino\":
```

```
{\"aeroporto\": \"BSB\",\"chegada\": {\"$date\":
1695482734000}},\"companhiaAerea\": \"AZUL\",\"preco\": {\"adulto\":
\"1000.00\",\"crianca\": \"500.00\"},\"_class\":
\"br.com.alura.mscompanhiasaereas.dominio.Voo\"}",
                "updateDescription": null,
                "source": {
                    "version": "2.3.0.Final",
                    "connector": "mongodb",
                    "name": "ms-companhias-aereas",
                    "ts_ms": 1695385534000,
                    "snapshot": "false",
                    "db": "livro-kafka",
                    "sequence": null,
                    "rs": "rs0",
                    "collection": "voos",
                    "ord": 1,
                    "lsid": null,
                    "txnNumber": null,
                    "wallTime": null
                },
                "op": "c",
                "ts_ms": 1695385534447,
                "transaction": null
            }
        },
        "encoding": "json",
        "schemaId": 0
    }
}
```


Além disso, é possível notar que o valor correspondente ao novo voo está em um formato JSON, mas aninhado, e não como o restante dos demais dados. E alguns consumidores podem ter dificuldade de ler informações desta forma:


```
"payload": {
    "after": "{\"_id\": {\"$oid\":
\"650d87bee0c6ff4b1a16a2a9\"},\"origem\": {\"aeroporto\":
\"SDU\",\"partida\": {\"$date\": 1695471934000}},\"destino\":
```

```
{\"aeroporto\": \"BSB\",\"chegada\": {\"$date\":
1695482734000}},\"companhiaAerea\": \"AZUL\",\"preco\": {\"adulto\":
\"1000.00\",\"crianca\": \"500.00\"},\"_class\":
\"br.com.alura.mscompanhiasaereas.dominio.Voo\"}",
}
```

Poderíamos preparar o *ms-pesquisa* para ler a mensagem inteira e extrair o que fosse necessário dela. Mas assim, teríamos que preparar um possível *binder* de JSON nessa aplicação para ler inclusive o que não é necessário para ela.

Há também alguns recursos existentes nos *connector plugins*, que nos permitem preparar a mensagem com apenas o que for necessário para os nossos propósitos, como os que veremos a seguir.

5.1 Remoção de JSON Schema do conteúdo das mensagens

> As alterações de código citadas neste capítulo estão presentes na branch *cap-5*, acessível via git. Para acessá-las, rodamos o comando a seguir na raiz do projeto:
>
> ```
> git checkout cap-5
> ```

No capítulo 3, vimos que era possível removermos o JSON Schema da chave da mensagem, usando a seguinte configuração no conector que criamos:

```
"key.converter.schemas.enable": "false",
```

Podemos também realizar o mesmo para o valor das mensagens, inserindo uma configuração muito semelhante à anterior:

```
"value.converter.schemas.enable": "false",
```

Essa configuração foi inserida no arquivo `docker/cdc-kafka-connect/novos-voos-conector-atualizar.json`. Repare que ele não possui o elemento `config` presente em `/docker/cdc-kafka-connect/novos-voos-conector.json`, visto que ele deve ser usado apenas para atualizar as configurações já existentes.

Precisamos informar também que queremos usar o `JsonConverter` na mensagem. Para isso, usamos a configuração a seguir:

```
"value.converter": "org.apache.kafka.connect.json.JsonConverter",
```

A atualização é feita usando a API REST do Kafka Connect, vista no capítulo 3. O *script* `novos-voos-conector-atualizar.sh` contém o comando necessário para realizar essa alteração:

```
curl -X PUT \
  http://localhost:8083/connectors/novos-voos-conector/config \
  -H 'content-type: application/json' \
  -d @novos-voos-conector-atualizar.json
```

Ao rodar esse comando, o Kafka Connect retorna uma mensagem semelhante à de criação de conectores:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo cdc-kafka-connect % ./novos-
voos-conector-atualizar.sh
{"name":"novos-voos-conector","config":
{"connector.class":"io.debezium.connector.mongodb.MongoDbConnector","mongo
db.connection.string":"mongodb://mongodb:27017","mongodb.user":"user","mon
godb.password":"password","database.include.list":"livro-
kafka","collection.include.list":"livro-
kafka.voos","key.converter":"org.apache.kafka.connect.json.JsonConverter",
"key.converter.schemas.enable":"false","skipped.operations":"u,t","topic.p
refix":"ms-companhias-
aereas","value.converter.schemas.enable":"false","name":"novos-voos-
```

conector"},"tasks":[{"connector":"novos-voos-
conector","task":0}],"type":"source"}%

E, ao enviarmos um novo voo, podemos notar que a mensagem diminuiu bastante de tamanho, embora contenha informações além do simples voo que acabamos de cadastrar:


```
{
    "partitionID": 0,
    "offset": 15,
    "timestamp": 1695391668592,
    "compression": "uncompressed",
    "isTransactional": false,
    "headers": [],
    "key": {
        "payload": {
            "id": "{\"$oid\": \"650d9fb3a4928f620b8acca4\"}"
        },
        "encoding": "json",
        "schemaId": 0
    },
    "value": {
        "payload": {
            "before": null,
            "after": "{\"_id\": {\"$oid\":
\"650d9fb3a4928f620b8acca4\"},\"origem\": {\"aeroporto\":
\"SDU\",\"partida\": {\"$date\": 1695478067000}},\"destino\":
{\"aeroporto\": \"BSB\",\"chegada\": {\"$date\":
1695488867000}},\"companhiaAerea\": \"AZUL\",\"preco\": {\"adulto\":
\"1000.00\",\"crianca\": \"500.00\"},\"_class\":
\"br.com.alura.mscompanhiasaereas.dominio.Voo\"}",
            "updateDescription": null,
            "source": {
                "version": "2.3.0.Final",
                "connector": "mongodb",
                "name": "ms-companhias-aereas",
                "ts_ms": 1695391668000,
                "snapshot": "false",
                "db": "livro-kafka",
```

```
                "sequence": null,
                "rs": "rs0",
                "collection": "voos",
                "ord": 1,
                "lsid": null,
                "txnNumber": null,
                "wallTime": null
            },
            "op": "c",
            "ts_ms": 1695391668306,
            "transaction": null
        },
        "encoding": "json",
        "schemaId": 0
    }
}
```


Vamos ver então como corrigir esse problema definitivamente usando *Single Message Transformations*.

5.2 Single Message Transformations (SMT)

Os SMTs são classes Java que, uma vez configuradas no conector, realizam alterações nas mensagens antes de o Kafka Connect enviá-las ao tópico. Tanto o Kafka Connect quanto o Debezium oferecem diversos SMTs. Eles podem realizar desde modificações simples nas mensagens até implementar padrões de roteamento, como *Content-Based Router*, onde o tópico que receberá a mensagem é determinado com base em campos da mensagem que o Kafka Connect escutar.

Mais informações sobre o padrão Content-Based Router podem ser encontradas em https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html ou no livro *Enterprise Integration Patterns* (HOHPE, G.; WOOLF, B., 2003)

Para trabalharmos com apenas o *payload.after* da mensagem (ou seja, com o voo) precisamos inserir o SMT `ExtractNewRecordState` do pacote `io.debezium.connector.mongodb.transforms` no nosso conector. Também precisamos dar um nome para a transformação (aqui a chamamos de `extractNewDocumentState`). Por último, precisamos informar que os *tombstone events* enviados pelo Kafka Connect devem ser enviados (via configuração `"transforms.unwrap.drop.tombstones": "false"`).

```
"transforms": "extractNewDocumentState",
"transforms.extractNewDocumentState.type":"io.debezium.connector.mongodb.transforms.ExtractNewDocumentState",
"transforms.extractNewDocumentState.drop.tombstones": "false"
```

Podemos inserir essas configurações no final do arquivo de atualização do conector `docker/cdc-kafka-connect/novos-voos-conector-atualizar.json`. Com o que já foi feito neste capítulo, ele ficará assim:

```
{
   "connector.class": "io.debezium.connector.mongodb.MongoDbConnector",
   "mongodb.connection.string": "mongodb://mongodb:27017",
   "mongodb.user": "user",
   "mongodb.password": "password",
   "database.include.list": "livro-kafka",
   "collection.include.list": "livro-kafka.voos",
   "key.converter": "org.apache.kafka.connect.json.JsonConverter",
   "key.converter.schemas.enable": "false",
   "skipped.operations": "u,t",
   "topic.prefix": "ms-companhias-aereas",
   "value.converter": "org.apache.kafka.connect.json.JsonConverter",
   "value.converter.schemas.enable": "false",
```

```
   "transforms": "extractNewDocumentState",
   "transforms.extractNewDocumentState.type":
"io.debezium.connector.mongodb.transforms.ExtractNewDocumentState",
   "transforms.extractNewDocumentState.drop.tombstones": "false"
}
```

O `ExtractNewDocumentState` está presente na biblioteca *debezium-scripting*, e essa precisa estar no nosso Kafka Connect. Então precisamos inserir o código a seguir no `Dockerfile` de criação do `cdc-kafka-connect`:

```
RUN echo ">>>>>>>>>> Instalando Bibliotecas de SMT <<<<<<<<<<"
RUN wget https://repo1.maven.org/maven2/io/debezium/debezium-
scripting/2.3.0.Final/debezium-scripting-2.3.0.Final.tar.gz
RUN tar xvfz debezium-scripting-2.3.0.Final.tar.gz --directory
$KAFKA_CONNECT_PLUGINS_DIR
```

> Note a versão da biblioteca, que é a mesma versão do Debezium.

Após essa alteração, devemos:

- parar a infra da solução, rodando o script `docker/infra-down.sh`;
- recriar a imagem `cdc-kafka-connect`, via script `docker/cdc-kafka-connect/dockerizar-kafka-connect.sh`;
- subir a infra da solução novamente, por meio de `docker/infra-up.sh`;
- atualizar o conector, usando `docker/cdc-kafka-connect/novos-voos-conector-atualizar.sh`;
- inserir um novo voo.

Após esses passos, podemos ver que a mensagem no tópico contém apenas as informações do voo:

```
{
    "partitionID": 0,
    "offset": 22,
    "timestamp": 1695397111915,
```

```
    "compression": "uncompressed",
    "isTransactional": false,
    "headers": [],
    "key": {
        "payload": {
            "id": "650db4f7e718646cb164a6ee"
        },
        "encoding": "json",
        "schemaId": 0
    },
    "value": {
        "payload": {
            "_id": "650db4f7e718646cb164a6ee",
            "origem": {
                "aeroporto": "SDU",
                "partida": 1695483510000
            },
            "destino": {
                "aeroporto": "BSB",
                "chegada": 1695494310000
            },
            "companhiaAerea": "AZUL",
            "preco": {
                "adulto": "1000.00",
                "crianca": "500.00"
            },
            "_class": "br.com.alura.mscompanhiasaereas.dominio.Voo"
        },
        "encoding": "json",
        "schemaId": 0
    }
}
```

A seguir, *links* para as listas de todos os SMTs. Vale a pena dar uma olhada:

- Kafka Connect: https://docs.confluent.io/platform/current/connect/transforms/overview.html
- Debezium: https://debezium.io/documentation/reference/stable/transformations/index.html

5.3 Recepção de voos via @KafkaListener, @KafkaHandler e o KafkaNull

Com as funcionalidades de criação e remoção de voos implementadas no *ms-companhias-aereas* e o envio de mensagens para o Kafka otimizado, vamos ver como ficou o ouvinte de eventos do *ms-pesquisa*.

Vale lembrar que esse ouvinte de eventos deve receber dois tipos de mensagem:

- mensagem de novo voo cadastrado: a mensagem contém tanto o a chave quanto o valor da mensagem. E a chave é o id do voo no banco de dados do *ms-companhias-aereas*;
- mensagem de voo cancelado: nesse caso, a mensagem contém apenas a chave que, assim como no caso anterior, é o id do voo. Entretanto, o valor da mensagem é nulo, graças aos *tombstone events* que vimos em capítulos anteriores.

Para que seja possível trabalharmos com mensagens de tipos diferentes vindas do mesmo tópico, o Spring Kafka permite que configuremos uma classe que contenha dois métodos, um para cada tipo de mensagem. A classe `ReceptorDeVoos` (pacote `br.com.alura.mspesquisa.infra`) do *ms-pesquisa* é a classe escolhida para receber esses dois métodos.

Ela contém, logo no início, anotações que a configuram como a receptora de voos:

```
@KafkaListener(topics = "${topico.voos.nome}",
        groupId = "${spring.kafka.group_id}",
        containerFactory = "voosKafkaListener")
```

O *container factory* configurado nessa última anotação foi criado na classe `KafkaObjects` (pacote `br.com.alura.mspesquisa.infra`):

```
@Bean
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,
VooCriadoEvent>> voosKafkaListener() {
    ConcurrentKafkaListenerContainerFactory<String, VooCriadoEvent>
factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(new DefaultKafkaConsumerFactory<String,
VooCriadoEvent>(
            consumerConfigs(VooCriadoEvent.class),
            new StringDeserializer(),
            jsonDeserializer(VooCriadoEvent.class)));
    return factory;
}
```

> Assim como no *ms-companhias-aereas*, todos os objetos que são usados pelo Kafka são configurados em uma classe `KafkaObjects`, também presente no *ms-pesquisa*.

Além disso, no `ReceptorDeVoos` é possível encontrarmos os métodos que recebem as mensagens. Ambos possuem a anotação `@KafkaHandler`, que informa ao Spring Kafka que o método é

responsável por receber mensagens do tópico. Cada método possui um conjunto de parâmetros específico:

```
package br.com.alura.mspesquisa.infra;

import br.com.alura.mspesquisa.infra.event.VooCriadoEvent;
import org.springframework.kafka.annotation.KafkaHandler;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.kafka.support.KafkaNull;
import org.springframework.messaging.handler.annotation.Header;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;

@Component
@KafkaListener(topics = "${topico.voos.nome}",
        groupId = "${spring.kafka.group_id}",
        containerFactory = "voosKafkaListener")
public class ReceptorDeVoos {

    @KafkaHandler
    void novoVoo(VooCriadoEvent novoVoo) {
        System.out.println("Voo recebido: " + novoVoo);
    }

    @KafkaHandler
    void vooCancelado(@Header(KafkaHeaders.RECEIVED_KEY) String idVoo,
                      @Payload(required = false) KafkaNull valorNulo) {
        System.out.println("Voo cancelado: " + idVoo);
    }
}
```

- o método `novoVoo()` recebe um objeto do tipo `VooCriadoEvent`, que contém a estrutura da mensagem de novo voo, recebida do tópico;
- o método `vooCancelado()` recebe 2 parâmetros: um que representa o *id* do voo a ser cancelado (configurado com a anotação `@Header(KafkaHeaders.RECEIVED_KEY)`, que informa ao Spring Kafka que esse parâmetro deve receber a chave da

mensagem) e o valor da mensagem, que é do tipo `KafkaNull` (e recebe a anotação `@Payload`, que informa ao Spring Kafka que o parâmetro deve receber o valor da mensagem).

`KafkaNull` é uma classe especial dentro do ecossistema Spring Kafka, e deve ser usada quando precisamos representar um valor nulo em mensagens Kafka. Ou seja, exatamente o que precisamos no caso de tratar mensagens de voos cancelados.

Como não precisamos do valor da mensagem quando um voo é cancelado, poderíamos simplesmente ignorar a presença de um valor na mensagem e não inserir o segundo parâmetro do método. Entretanto, se fizermos isso, o Spring Kafka terá dificuldades de saber qual dos métodos devem ser usados quando mensagens de cancelamento de voos forem disparadas, e lançará uma exceção semelhante à seguinte:

```
org.springframework.kafka.listener.ListenerExecutionFailedException:
Listener failed
    at
org.springframework.kafka.listener.KafkaMessageListenerContainer$Liste
nerConsumer.decorateException(KafkaMessageListenerContainer.java:2946)
~[spring-kafka-3.0.10.jar:3.0.10]
    at
org.springframework.kafka.listener.KafkaMessageListenerContainer$Liste
nerConsumer.doInvokeOnMessage(KafkaMessageListenerContainer.java:2887)
~[spring-kafka-3.0.10.jar:3.0.10]
    at
org.springframework.kafka.listener.KafkaMessageListenerContainer$Liste
nerConsumer.invokeOnMessage(KafkaMessageListenerContainer.java:2854) ~
[spring-kafka-3.0.10.jar:3.0.10]
    ...
    ...
Caused by: org.springframework.kafka.KafkaException: No method found
for class org.springframework.kafka.support.KafkaNull
    at
org.springframework.kafka.listener.adapter.DelegatingInvocableHandler.
getHandlerForPayload(DelegatingInvocableHandler.java:183) ~[spring-
kafka-3.0.10.jar:3.0.10]
    at
org.springframework.kafka.listener.adapter.DelegatingInvocableHandler.
getMethodNameFor(DelegatingInvocableHandler.java:311) ~[spring-kafka-
```

```
3.0.10.jar:3.0.10]
    at
org.springframework.kafka.listener.adapter.HandlerAdapter.getMethodAsS
tring(HandlerAdapter.java:75) ~[spring-kafka-3.0.10.jar:3.0.10]
    at
org.springframework.kafka.listener.adapter.MessagingMessageListenerAda
pter.invokeHandler(MessagingMessageListenerAdapter.java:391) ~[spring-
kafka-3.0.10.jar:3.0.10]
    ...
    ...
```

5.4 Resumo

Neste capítulo, vimos como deixar a mensagem enviada para o tópico com apenas as informações que precisamos, usando uma das SMTs disponíveis. Também começamos a entender como o *ms-pesquisa* deve ser implementado.

No próximo capítulo, veremos as demais classes pertencentes ao *ms-pesquisa*. Também veremos a integração do Kafka Connect com a base de dados MySQL, usada por esse microsserviço.

Capítulo 6
Busca e seleção de voos no ms-pesquisa

No capítulo anterior, o *ms-pesquisa* passou a receber os voos provenientes do *ms-companhias-aereas*. Com isso, ele poderá listar os voos para nossos viajantes, que poderão consultar essa lista e selecionar o voo desejado. Entretanto, o *ms-pesquisa* ainda não os persiste em banco, nem mesmo os remove quando necessário.

> Para acessar as alterações presentes no código para este capítulo, rodamos o comando abaixo na raiz do projeto:
>
> `git checkout cap-6`

Para que os voos possam ser persistidos (e futuramente consultados) vamos inserir o MySQL 8 no `docker-compose.yml`, com uma configuração bem simples, sem senha para usuário de banco inclusive:

```
mysql:
    image: mysql:8.0.23
    hostname: mysql
    container_name: mysql
    restart: always
    ports:
        - "3306:3306"
    environment:
        MYSQL_DATABASE: pesquisa
        MYSQL_ALLOW_EMPTY_PASSWORD: "yes"
    networks:
        bubble:
```

Com isso, podemos complementar nosso diagrama com a inclusão do novo banco de dados:

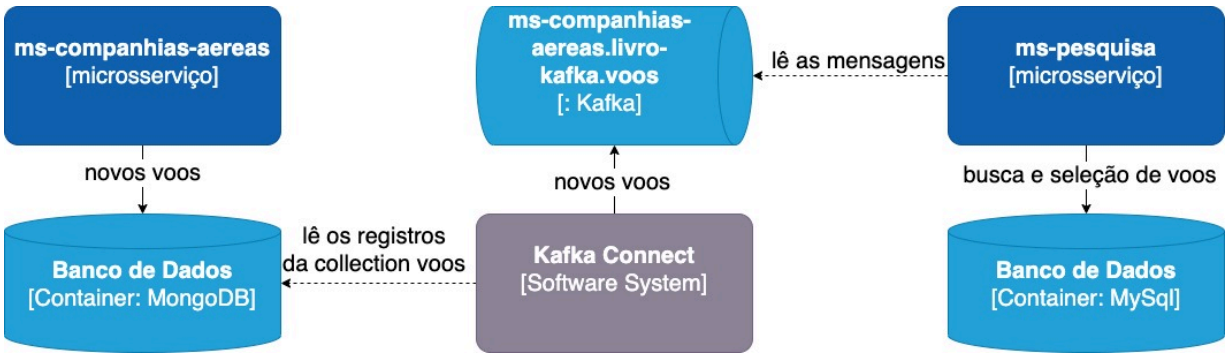


Figura 6.1: Adição da base MySQL ao ms-pesquisa.

Precisamos também inserir uma interface de repositório com o banco. Essa foi inserida no *listener* escrito no capítulo anterior. Dessa forma, a persistência de novos voos ou mesmo a sua remoção do banco de dados serão possíveis.

```
@Component
@KafkaListener(topics = "${topico.voos.nome}",
        groupId = "${spring.kafka.group_id}",
        containerFactory = "voosKafkaListener")
public class ReceptorDeVoos {

    private RepositorioDeNovosVoos repositorioDeNovosVoos;

    @Autowired
    ReceptorDeVoos(RepositorioDeNovosVoos repositorioDeNovosVoos) {
        this.repositorioDeNovosVoos = repositorioDeNovosVoos;
    }

    @KafkaHandler
    void novoVoo(VooCriadoEvent vooCriadoEvent) {
        System.out.println("Voo recebido: " + vooCriadoEvent);
        repositorioDeNovosVoos.save(vooCriadoEvent.toVoo());
    }

    @KafkaHandler
    void vooCancelado(@Header(KafkaHeaders.RECEIVED_KEY) String idVoo,
                      @Payload(required = false) KafkaNull valorNulo) {
        var identificadorCompanhiaAerea =
```

```
IdentificadorCompanhiaAerea.fromJson(idVoo);
        System.out.println("Voo cancelado: " +
identificadorCompanhiaAerea);

repositorioDeNovosVoos.deleteByIdCompanhiaAerea(identificadorCompanhiaAere
a);
    }
}
```

Além do *listener* de mensagens, o *ms-pesquisa* deve conter de alguns *endpoints*:

- busca de voos
- seleção de um voo específico

Neste capítulo, implementaremos esses *endpoints*, além de vermos a integração do banco de dados MySQL com o Kafka Connect.

6.1 Busca de voos

De forma simples, a busca de voos precisa dos seguintes parâmetros para ocorrer:

- aeroporto de origem
- data de partida
- aeroporto de destino

Essa busca pode ser realizada usando o comando cURL a seguir:

```
curl -X GET 'http://localhost:8081/voos?nome_origem=CGH&data_origem=2024-
08-21&nome_destino=SDU'
```

Assim como vimos no capítulo 3, essa busca pode também ser realizada via Postman (acessando o arquivo `livro-kafka.postman_collection.json`), o que facilita alterações nos parâmetros de busca.

Para facilitar os testes, assim que o *ms-pesquisa* inicia, são carregados 10 voos iniciais. Não é necessário cadastrar voos no *ms-companhias-aereas* para que os testes dessa aplicação sejam possíveis.

A busca citada é baseada em um voo pré-cadastrado no sistema, e retorna o seguinte voo:


```
[
    {
        "idCompanhiaAereaIdentificador": "65136e3571e71b6bcc8b3028",
        "origemAeroporto": "CGH",
        "origemPartida": "2024-08-21T10:38:39",
        "destinoAeroporto": "SDU",
        "destinoChegada": "2024-08-21T12:38:39",
        "companhiaAerea": "GOL",
        "precoAdulto": 1000.00,
        "precoCrianca": 500.00
    },
    {
        "idCompanhiaAereaIdentificador": "65136e3571e71b6bcc8b3027",
        "origemAeroporto": "CGH",
        "origemPartida": "2024-08-21T18:38:39",
        "destinoAeroporto": "SDU",
        "destinoChegada": "2024-08-21T20:38:39",
        "companhiaAerea": "AZUL",
        "precoAdulto": 1000.00,
        "precoCrianca": 500.00
    }
]
```


6.2 Seleção de voo

Com a lista de todos os voos refinada, o viajante pode selecionar o voo desejado. E, assim que essa seleção é feita, ele usa o serviço de

seleção de voo para informar sua escolha ao sistema.

Nessa seleção, ele deve informar o identificador do voo na companhia aérea, a quantidade de adultos e crianças que devem viajar, bem como seu CPF (para que seja possível identificar o comprador). Essas informações são enviadas ao *ms-pesquisa* por meio da chamada a seguir:

```
curl -X POST 'http://localhost:8081/voos-selecionados' \
--header 'Content-Type: application/json' \
--data '{
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3027",
    "adultos": 1,
    "criancas": 2,
    "cpfComprador": "75625404704"
}'
```

De posse do `idCompanhiaAerea` é possível obtermos o objeto `Voo`, presente em banco. A partir do `Voo` encontrado, podemos calcular os valores da viagem a ser feita, de acordo com os passageiros que estarão nela. Todas essas informações nos permitem criar o objeto `VooSelecionado` a seguir:

```
@Entity
@Table(name = "voos_selecionados")
@EqualsAndHashCode(exclude = "id")
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class VooSelecionado {

    @Id
    @GeneratedValue
    private Long id;

    private int adultos;
    private int criancas;

    @Embedded
    private IdentificadorCompanhiaAerea identificadorCompanhiaAerea;
```

```
    @Enumerated(EnumType.STRING)
    private Aeroporto destino;

    private BigDecimal totalAdultos;
    private BigDecimal totalCriancas;
    private BigDecimal total;

    @Embedded
    private CpfComprador cpfComprador;

    public VooSelecionado(Voo voo, int adultos, int criancas, CpfComprador
cpfComprador){
        this.identificadorCompanhiaAerea = voo.getIdCompanhiaAerea();
        this.adultos = adultos;
        this.criancas = criancas;
        this.destino = voo.destino();
        this.totalAdultos = voo.calcularTotalAdultos(adultos);
        this.totalCriancas = voo.calcularTotalCriancas(criancas);
        this.total = totalAdultos.add(totalCriancas);
        this.cpfComprador = cpfComprador;
    }

    public IdentificadorCompanhiaAerea idCompanhiaAerea() {
        return identificadorCompanhiaAerea;
    }
}
```

Esses voos selecionados deverão ser enviados para um tópico Kafka, para que o *ms-selecao-hospedagens* possa selecionar hospedagens próximas ao destino do(s) viajante(s).

6.3 Integração entre o ms-pesquisa e o ms-selecao-hospedagens via Kafka Connect

De forma semelhante ao *ms-pesquisa*, o *ms-selecao-hospedagens* também terá dois *endpoints*:

- um permitirá ao viajante selecionar uma hospedagem próxima ao local de destino;
- o outro fará a seleção efetiva da hospedagem.

> O *ms-selecao-hospedagens* será implementado no capítulo 7.

Para que esses endpoints funcionem, precisamos implementar um conector ligando a tabela `voos_selecionados` com um tópico Kafka.

E, para instalarmos esse conector no Kafka Connect, seguimos os mesmos passos de instalação do conector com o MongoDB:

- inicialmente, precisamos atualizar o `Dockerfile` do nosso *cdc-kafka-connect*, com a inclusão do *connector plugin* para MySQL:

```
RUN echo ">>>>>>>>>> Instalando Plugin do mysql <<<<<<<<<<"
RUN wget  https://repo1.maven.org/maven2/io/debezium/debezium-connector-mysql/2.4.0.Final/debezium-connector-mysql-2.4.0.Final-plugin.tar.gz
RUN tar xvfz debezium-connector-mysql-2.4.0.Final-plugin.tar.gz --directory $KAFKA_CONNECT_PLUGINS_DIR
```

- depois, geramos a imagem do *cdc-kafka-connect* novamente, usando o *script* `dockerizar-kafka-connect.sh`, presente na pasta `docker/cdc-kafka-connect`;
- com a imagem do Kafka Connect atualizada, podemos rodar nossa infra de novo, usando o *script* `infra-up.sh`;
- por último, podemos rodar o *ms-pesquisa*, executando o comando `mvn -f ms-pesquisa/pom.xml compile spring-boot:start` na raiz do projeto.

Assim que o *ms-pesquisa* começar a rodar, será possível notar, no *log* da aplicação, a carga inicial do banco entrando em ação, inserindo diversos voos iniciais.

Serão vistas diversas linhas semelhantes à seguinte:

```
Hibernate: select
v1_0.id,v1_0.companhia_aerea,v1_0.desembarque,v1_0.chegada,v1_0.identificador,v1_0.embarque,v1_0.partida,v1_0.adulto,v1_0.crianca from voos v1_0
where v1_0.identificador=?
```

O conector entre a tabela `voos_selecionados` e o tópico Kafka possui diversas semelhanças com o conector usado no MongoDB:

- também usamos uma estrutura JSON com `name` e `config`;
- precisamos indicar a classe do conector, que nesse caso é o `io.debezium.connector.mysql.MySqlConnector`;
- indicamos o servidor de banco de dados por meio das configurações `database.hostname`, `database.port`, `database.user` e `database.password` (este último não foi usado no nosso caso, visto que nosso banco foi configurado para ser usado sem senha;
- a configuração `database.include.list` informa o(s) banco(s) de dados que deve(m) ser lido(s). No nosso caso, apenas o banco de dados `pesquisa` deve ser lido;
- a configuração `table.include.list` informa qual(is) tabela(s) a ser(em) lida(s). Embora possa parecer redundante, esse valor deve ser informado no formato *nomeDoBanco.nomeDaTabela* (ou seja, com o nome do banco de dados ao qual a tabela pertence). Logo, preenchemos essa configuração com o valor `pesquisa.voos_selecionados`;
- informamos quais operações DML (*Data Manipulation Language*) não precisam ser lidas, usando a configuração `skipped.operations`;
- `topic.prefix`: o nome que identifica a fonte de dados para o Kafka Connect.

Há também alguns campos novos, se compararmos com as configurações usadas pelo conector com o MongoDB:

- `schema.history.internal.kafka.bootstrap.servers`: armazena todas as mudanças que ocorreram no `schema` do banco de dados lidas pelo conector, em uma instância de Kafka. Dessa forma, é

possível fazermos uma auditoria nas alterações realizadas no banco de dados ao longo do tempo. Também fornece uma maneira de fazer com que os consumidores fiquem informados quanto a possíveis alterações no formato da mensagem.

- `schema.history.internal.kafka.topic` : determina qual será o nome do tópico que armazenará as informações citadas acima.

O conector completo encontra-se a seguir:

```
{
    "name": "voo-selecionado-conector",
    "config": {
        "connector.class": "io.debezium.connector.mysql.MySqlConnector",
        "database.hostname": "mysql",
        "database.port": "3306",
        "database.user": "root",
        "database.server.id": "1234",
        "database.include.list": "pesquisa",
        "table.include.list": "pesquisa.voos_selecionados",
        "skipped.operations": "u,d,t",
        "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
        "schema.history.internal.kafka.topic": "ms-pesquisa-
schemahistory",
        "topic.prefix": "ms-pesquisa"
    }
}
```

Para inserirmos esse conector no Kafka Connect, basta que rodemos o comando `./voo-selecionado-conector.sh` presente em `docker/cdc-kafka-connect`.

Com esse conector instalado, o diagrama da solução passa a ser da seguinte forma:

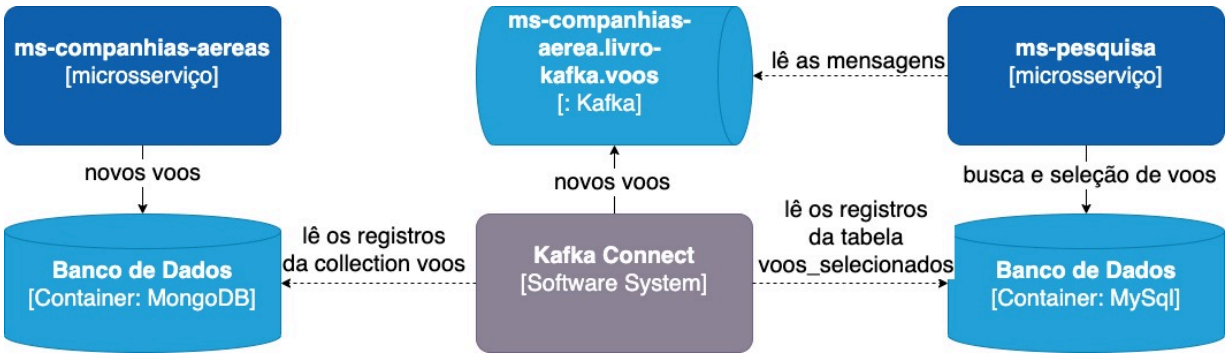


Figura 6.2: Adição do conector entre a base do ms-pesquisa e o Kafka Connect.

6.4 Publicação de mudanças no schema do banco de dados por padrão

Entretanto, ao instalarmos esse conector, diversas linhas como a seguinte aparecerão no log do Kafka Connect:

```
[2023-10-14 12:03:13,357] WARN [Producer clientId=connector-producer-voo-selecionado-conector-0] Error while fetching metadata with correlation id 5 : {ms-pesquisa=UNKNOWN_TOPIC_OR_PARTITION} (org.apache.kafka.clients.NetworkClient)
```

Mensagens como essa aparecem quando tópicos que não existem estão sendo buscados pelo Kafka Connect.

E, de fato, se analisarmos o `docker-compose.yml`, veremos que o tópico `ms-pesquisa` não está na lista de tópicos a serem criados:

```
criador-topicos:
    image: confluentinc/cp-kafka:7.3.2
    container_name: criador-topicos
    depends_on:
        - kafka
    restart: on-failure
    command:
        - bash
```

```
        - -c
        - |
          echo -e 'Aguardando Kafka subir com sucesso...'
          kafka-topics --bootstrap-server kafka:29092 --list

          kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic ms-companhias-aereas.livro-
kafka.voos
          kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic ms-
pesquisa.pesquisa.voos_selecionados

          echo -e 'Criados os seguintes topicos com sucesso:'
          kafka-topics --bootstrap-server kafka:29092 --list
    networks:
        bubble:
```

Podemos criar o tópico `ms-pesquisa` manualmente, acessando o terminal do contêiner do Kafka:

```
rodrigovieirapinto@MBP-de-Rodrigo ~ % docker exec -it kafka bash
[appuser@kafka ~]$ kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic ms-pesquisa
Created topic ms-pesquisa.
```

Assim que criamos o tópico, vimos que o log informa que diversos registros foram enviados para o tópico recém-criado:

```
[2023-10-14 12:40:56,175] INFO 10 records sent during previous
00:00:37.279, last recorded offset of {server=ms-pesquisa} partition is
{ts_sec=1697287220, file=binlog.000002, pos=5732}
(io.debezium.connector.common.BaseSourceTask)
```

E se acessarmos o tópico no Redpanda, veremos diversas mensagens semelhantes à seguinte:

```
{
   "schema":{
      "type":"struct"
      "fields":[...]
```

```
      "optional":false
      "name":"io.debezium.connector.mysql.SchemaChangeValue"
      "version":1
   }
   "payload":{
      "source":{...}
      "ts_ms":1698591810344
      "databaseName":"pesquisa"
      "schemaName":NULL
      "ddl":"create table voos (id bigint not null auto_increment,
companhia_aerea enum ('AVIANCA','AZUL','GOL','LATAM'), desembarque enum
('BSB','CGH','GRU','SDU'), chegada datetime(6), identificador
varchar(255), embarque enum ('BSB','CGH','GRU','SDU'), partida
datetime(6), adulto decimal(38,2), crianca decimal(38,2), primary key
(id)) engine=InnoDB"
      "tableChanges":[
         0:{
            "type":"CREATE"
            "id":""pesquisa"."voos""
            "table":{
               "defaultCharsetName":"utf8mb4"
               "primaryKeyColumnNames":[...]
               "columns":[
                  0:{...}
                  1:{...}
                  2:{...}
                  3:{...}
                  4:{...}
                  5:{...}
                  6:{...}
                  7:{...}
                  8:{...}
               ]
               "comment":NULL
            }
         }
      ]
   }
}
```

Observação: alguns trechos da mensagem acima foram omitidos por brevidade.

No entanto, se acessarmos o tópico `ms-pesquisa-schemahistory` encontraremos mensagens no mesmo formato acima, com as mesmas informações. Por que o conector armazena informações iguais em tópicos diferentes?

Segundo a documentação do conector Debezium para MySQL (e dos conectores Debezium para diversas outras bases relacionais) o conector pode acessar informações referentes a comandos DDL (*Data Definition Language*) aplicados no banco acessando o tópico determinado pela configuração `schema.history.internal.kafka.topic`. Esse tópico deve ser de acesso exclusivo do Kafka Connect e do conector. Portanto, caso alguma aplicação também precise dessas informações, é necessário utilizar o tópico determinado pela configuração `topic.prefix`.

Mais informações podem ser encontradas em https://debezium.io/documentation/reference/stable/connectors/mysql.html

O *ms-selecao-hospedagens* não utilizará as informações relacionadas ao *schema* do banco de dados do *ms-pesquisa*. Logo, podemos fazer com que o conector deixe de alimentar o tópico `ms-pesquisa`. Para isso, podemos enviar uma atualização do conector recém-criado, no formato adiante:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
```

```
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false"
}
```

A atualização acima está no arquivo `voo-selecionado-conector-atualizar.json`, presente na pasta `docker/cdc-kafka-connect`. Para executar essa atualização, rode o *script* `voo-selecionado-conector-atualizar.sh`, na mesma pasta.

Repare que a configuração `include.schema.changes` foi incluída. Seu valor *default* é *true*. Configurada para *false*, o tópico `ms-pesquisa` deixará de ser usado, podendo inclusive ser removido. Assim podemos economizar espaço em disco não persistindo mensagens que não serão usadas por outras aplicações.

6.5 Formato das mensagens e o JsonConverter

Com esse ajuste, podemos voltar nossas atenções às mensagens enviadas ao tópico `ms-pesquisa.pesquisa.voos_selecionados`. O diagrama atualizado (contendo o tópico) encontra-se a seguir:

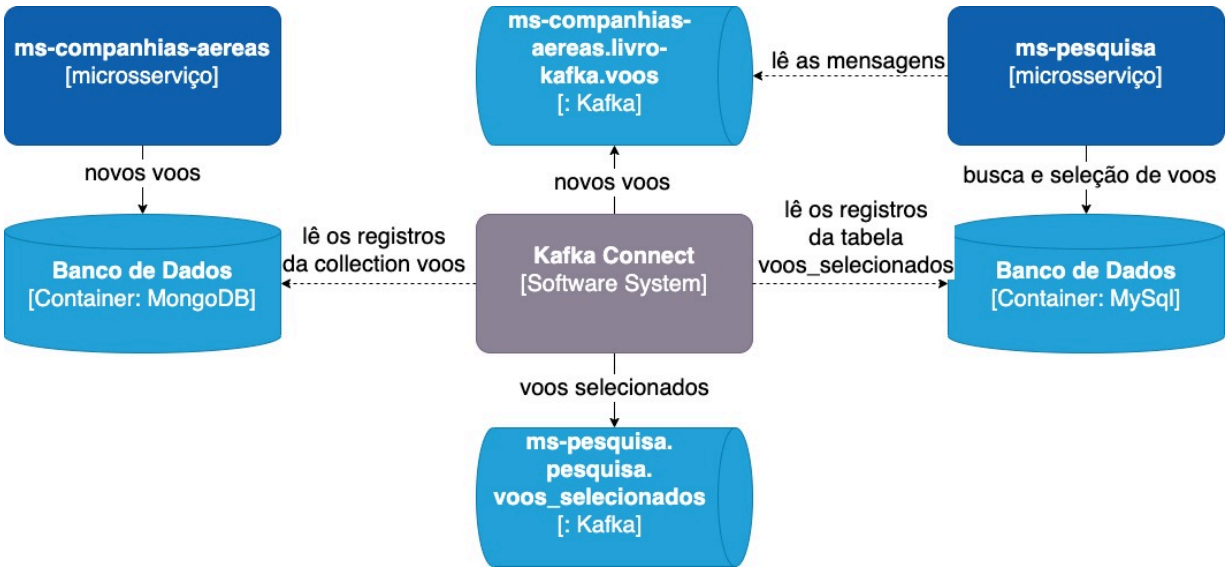


Figura 6.3: Adição de novo tópico, que receberá os voos selecionados.

Repare que, de forma semelhante ao *connector plugin* para MongoDB, o *connector plugin* para MySQL concatena informações do conector para formar o nome do tópico de destino das mensagens. No caso, foram utilizadas as configurações:

- `topic.prefix`
- `table.include.list`

Conforme já vimos, ao selecionarmos um voo:

- o voo selecionado é persistido em banco;
- o voo selecionado é enviado ao tópico acima, graças ao conector que instalamos.

Ao selecionarmos o voo, portanto, podemos ver uma mensagem semelhante à seguinte, no tópico de voos selecionados. Como não utilizamos nenhum SMT ainda, é possível vermos que ela possui diversos metadados:

```
"payload": {
    "before": null,
```

```
    "after": {
        "id": 1,
        "adultos": 1,
        "cpf": "75625404704",
        "criancas": 2,
        "destino": "SDU",
        "identificador": "65136e3571e71b6bcc8b3027",
        "total": "Aw1A",
        "total_adultos": "AYag",
        "total_criancas": "AYag"
    },
    "source": {
        "version": "2.4.0.Final",
        "connector": "mysql",
        "name": "ms-pesquisa",
        "ts_ms": 1697292634000,
        "snapshot": "false",
        "db": "pesquisa",
        "sequence": null,
        "table": "voos_selecionados",
        "server_id": 1,
        "gtid": null,
        "file": "binlog.000002",
        "pos": 6306,
        "row": 0,
        "thread": 20,
        "query": null
    },
    "op": "c",
    "ts_ms": 1697292634343,
    "transaction": null
}
```


Diversos trechos da mensagem foram omitidos por brevidade.

Assim como fizemos na mensagem oriunda do MongoDB, podemos também aplicar SMTs que façam com que apenas o `payload.after` seja enviado ao tópico. Para isso, podemos atualizar o conector

aplicando o SMT `ExtractNewRecordState` inserindo-o no *script* de atualização do conector:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "transforms": "extractNewDocumentState",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState"
}
```

> Como já instalamos as bibliotecas de SMT no capítulo anterior, não precisamos nos preocupar com sua instalação aqui.

Após a aplicação dessa alteração, a mensagem diminuiu de tamanho. Mas ainda contém informações além do *payload*:

```
{
   "schema":{
      "type":"struct"
      "fields":[...]
      "optional":false
    "name":"ms-pesquisa.pesquisa.voos_selecionados.Value"
   }
   "payload":{
      "id":8
      "adultos":1
      "cpf":"75625404704"
```

```
      "criancas":2
      "destino":"SDU"
      "identificador":"65136e3571e71b6bcc8b3027"
      "total":"2000.00"
      "total_adultos":"1000.00"
      "total_criancas":"1000.00"
    }
}
```

Para mantermos somente o *payload*, precisamos inserir um conversor de mensagem. No caso, usaremos o `JsonConverter`, já visto em capítulos anteriores. Também devemos informar que não queremos que o conversor envie o *schema* da mensagem.

Ao inserirmos essas configurações no *script* de atualização do conector, ele ficará desta forma:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState"
}
```

Ao selecionar outro voo, finalmente temos apenas as informações relacionadas a ele no tópico.

```
{
    "id":2,
    "adultos":1,
    "cpf":"75625404704",
    "criancas":2,
    "destino":"SDU",
    "identificador":"65136e3571e71b6bcc8b3027",
    "total":"Aw1A",
    "total_adultos":"AYag",
    "total_criancas":"AYag"
}
```


Mas temos outro problema para resolver: os valores totais foram persistidos no tópico como caracteres estranhos, e não como valores numéricos ou decimais.

Esse problema pode ocorrer quando o Kafka Connect precisa lidar com colunas do tipo DECIMAL ou NUMERIC. E, se verificarmos a estrutura da tabela `voos_selecionados`, veremos que o Spring Data JPA criou as colunas que armazenam valores monetários com o tipo DECIMAL:

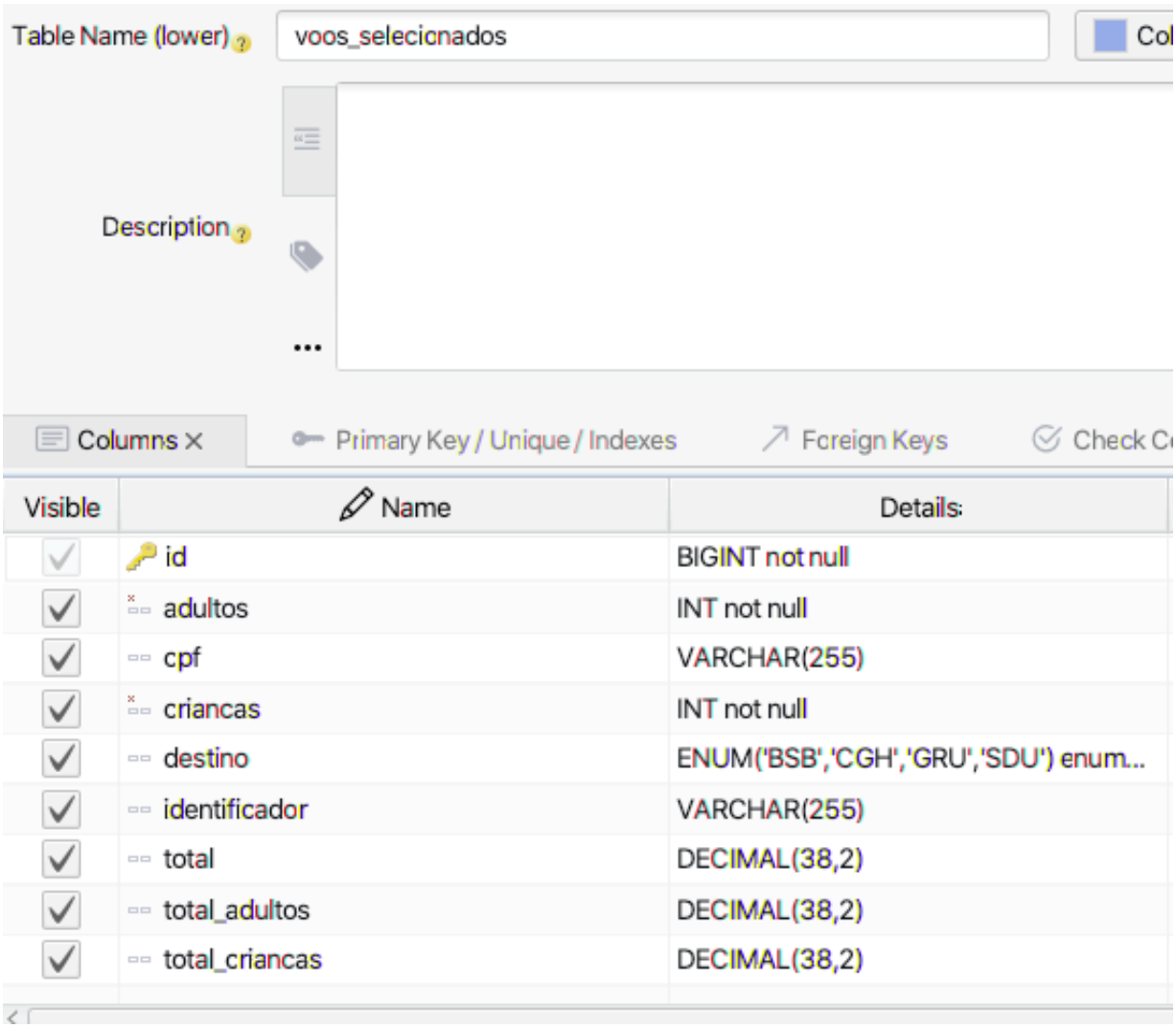


Figura 6.4: Estrutura da tabela voos_selecionados.

É necessário, portanto, configurar o conector para a maneira correta com que ele deve ler os dados presentes nessas colunas. A configuração que precisamos inserir no conector é `decimal.handling.mode`, que suporta 3 valores diferentes:

- `double`: configurada dessa forma, fará com que os valores sejam persistidos no tópico como valores do tipo `double`. Embora um valor dessa forma possa ser mais fácil de ser lido por diversos consumidores, há a possibilidade de perda de precisão, devido a problemas já conhecidos em conversões de `double` para `BigDecimal` e vice-versa;

- `precise` (*default*): armazena o valor num formato binário, mantendo inclusive o tipo;
- `string` : armazena o valor no formato `String` . Perde-se o tipo, mas mantém-se a precisão.

> Sobre os problemas relacionados a esse tipo de conversão: uma boa referência é https://www.alura.com.br/artigos/arredondamento-no-java-do-double-ao-bigdecimal

Para usarmos o valor *default* nessa configuração, seria importante garantir que todos os consumidores do tópico pudessem realizar conversões de binário para `BigDecimal` , e não temos essa garantia. Por outro lado, se configurarmos para `string` , será mais fácil para consumidores realizarem a conversão. Então, vamos atualizar o conector com essa nova configuração com o valor `string` :

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string"
}
```

Após a atualização do conector, os novos voos selecionados passam a exibir os valores decimais como `String`:

```
{
    "id":4,
    "adultos":1,
    "cpf":"75625404704",
    "criancas":2,
    "destino":"SDU",
    "identificador":"65136e3571e71b6bcc8b3027",
    "total":"2000.00",
    "total_adultos":"1000.00",
    "total_criancas":"1000.00"
}
```

6.6 Formato das chaves no tópico, e a concatenação de SMTs

Além das questões relacionadas ao formato da mensagem, as chaves (*key*) da mensagem estão sendo gravadas assim:

```
Struct{id=1}
```

onde o valor do campo `id` é o valor da chave primária da tabela `voos_selecionados`.

A informação de que a chave é uma `Struct` refere-se à maneira como as mensagens (e suas chaves) são tratadas pelo Kafka Connect. No que diz respeito a identificar unicamente um voo selecionado, essa informação não é importante, podendo ser suprimida.

Podemos então, manter apenas o valor do campo `id` como valor da chave de cada mensagem. Para fazermos isso, usaremos de outro SMT, o `ExtractField$Key`. Esse SMT, basicamente, lê o valor de um campo da mensagem e o configura como chave dela.

Inicialmente, vamos informar no *script* de atualização do conector que haverá um novo SMT. Vamos chamá-lo de `extractKey`. Para isso, precisamos atualizar a configuração `transforms` da seguinte forma:

```
"transforms": "extractNewDocumentState,extractKey",
```

Depois declaramos o novo SMT:

```
"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
```

Por último, informamos ao SMT qual campo ele deve considerar como sendo a nova chave:

```
"transforms.extractKey.field":"id"
```

O *script* atualizado ficará da seguinte forma:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
    "transforms.extractKey.field":"id",
```

```
    "decimal.handling.mode": "string"
}
```

> A ordem de declaração dos SMTs na configuração `transforms` é importante. Ela determina a sequência de aplicação deles nas mensagens.

6.7 Leitura de dados do MySQL pelos conectores CDC

Conforme vimos no capítulo 4, o Kafka Connect e os conectores do tipo CDC não leem as tabelas, mas os logs de banco de dados correspondentes a essas tabelas. E com o MySQL não é diferente.

Para realizar a leitura de dados do MySQL, é necessário basicamente que o **log binário do MySQL esteja habilitado**. Esse log contém todas as operações realizadas no banco de dados e, portanto, deve ser manuseado com cuidado, pois pode conter dados sensíveis.

Por padrão, o log binário está habilitado nas versões 8 e superiores do MySQL. Caso esteja trabalhando com uma versão inferior, pode ser necessário habilitá-lo. No MySQL 5.7, é necessário que o arquivo `/etc/mysql/conf.d/my-custom.cnf` esteja com as informações a seguir:

```
[mysqld]
skip-host-cache
skip-name-resolve
symbolic-links=0
server-id=1234
log_bin=mysql-bin
expire_logs_days=1
binlog_format=row
```

Para verificar se o log binário está habilitado, pode-se usar as *queries* a seguir:

```
-- versão 5.7
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE
variable_name='log_bin';

-- para versões 8 ou superiores
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM performance_schema.global_variables WHERE
variable_name='log_bin';
```

Por estar em um formato binário, é necessário o uso da ferramenta `mysqlbinlog` para que ele possa ser exibido em um formato legível por humanos.

Podemos verificar qual o arquivo `binlog` é o mais atual lendo a última mensagem do tópico `ms-pesquisa-schemahistory`. O nome do arquivo é um dos campos da mensagem:

```
{
    "source":{
        "server":"ms-pesquisa",
    },
    "position":{
        "ts_sec":1697307031,
        "file":"binlog.000002",
        "pos":5731,
        "snapshot":true,
    },
    "ts_ms":1697307031343,
    "databaseName":"pesquisa",
    "ddl":"CREATE TABLE `voos_selecionados_seq` ( `next_val` bigint
DEFAULT NULL ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci"
}
```


Pela mensagem, o nome do arquivo mais recente é o `binlog.000002`. Então, podemos acessar o contêiner do MySQL e verificar seu conteúdo:

```
rodrigovieirapinto@MBP-de-Rodrigo ~ % docker exec -it mysql bash
root@mysql:/# mysqlbinlog  -v /var/lib/mysql/binlog.000002
```

É possível verificar as inserções na tabela `voos_selecionados`. A seguir, veja um trecho do resultado da consulta que realizamos acima:


```
BEGIN
/*!*/;
# at 7389
#231014 18:32:10 server id 1  end_log_pos 7482 CRC32 0xbb226265
Table_map: `pesquisa`.`voos_selecionados` mapped to number 118
# at 7482
#231014 18:32:10 server id 1  end_log_pos 7627 CRC32 0xbbf95159
Write_rows: table id 118 flags: STMT_END_F

BINLOG '
qt4qZRMBAAAAXQAAADodAAAAAHYAAAAAAAEACHBlc3F1aXNhABF2b29zX3NlbGVjaW9uYWRvcw
AJ
CAMPA/4P9vb2DPwD9wH8AyYCJgImAvQBAQEAAgP8/wBlYiK7
```

```
qt4qZR4BAAAAkQAAAMsdAAAAAHYAAAAAAEAAgAJ//8AAAMAAAAAAAAAAQAAAAsANzU2MjU0MDQ3
MDQCAAAABBgANjUxMzZlMzU3MWU3MWI2YmNjOGIzMDI3gAAAAAAAAAAAAAAAAAH0ACAAAAAAAA
AAAAAAAAAAPoAIAAAAAAAAAAAAAAAAAA+gAWVH5uw==
'/*!*/;
### INSERT INTO `pesquisa`.`voos_selecionados`
### SET
###   @1=3
###   @2=1
###   @3='75625404704'
###   @4=2
###   @5=4
###   @6='65136e3571e71b6bcc8b3027'
###   @7=2000.00
###   @8=1000.00
###   @9=1000.00
# at 7627
#231014 18:32:10 server id 1  end_log_pos 7658 CRC32 0xc4086286     Xid =
361
COMMIT/*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
DELIMITER ;
# End of log file
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

6.8 Resumo

Neste capítulo, pudemos ver a integração do Kafka Connect com a base de dados do *ms-pesquisa*, que é MySQL. Pudemos ver também diversas maneiras de ajustar as mensagens conforme nossas necessidades, usando configurações do conector ou mesmo novas SMTs. Também pudemos ver como o Kafka Connect lê os logs do MySQL.

No próximo capítulo, faremos a integração do *ms-pesquisa* com o *ms-selecao-hospedagens*. Por meio dessa integração, conseguiremos entregar um pacote completo (com voo e hospedagem) para nosso viajante. Também veremos a integração do Kafka Connect com o banco de dados do *ms-selecao-hospedagens*: o Postgres.

Capítulo 7
Busca e seleção de hospedagens no ms-selecao-hospedagens

Agora que é possível selecionar voos na nossa solução, vamos implementar uma solução que permita também selecionar a hospedagem onde nossos viajantes poderão descansar, se necessário. Para que isso seja possível, implementaremos neste capítulo o *ms-selecao-hospedagens*, que terá uma estrutura muito semelhante ao *ms-pesquisa*, mas para hospedagens:

- implementaremos um *endpoint* para busca de hospedagens. Essa busca terá a sigla do aeroporto como único parâmetro de busca. E as hospedagens a serem exibidas serão as mais próximas dele. Assim, o deslocamento entre o aeroporto e o local onde o viajante ficará será menor e mais rápido;
- também escreveremos um *endpoint* para que o viajante marque uma hospedagem como selecionada. Essa seleção será persistida em banco, e será enviada também para um tópico, por meio do Kafka Connect e um conector. Esse tópico será consumido por aplicações que veremos nos capítulos seguintes.

O banco de dados que usaremos para esse sistema será o Postgres. Podemos inseri-lo no `docker-compose.yml` da maneira a seguir:

```
postgres:
    image: postgres:16
    hostname: postgres
    container_name: postgres
    restart: always
    environment:
        POSTGRES_DB: hospedagens
        POSTGRES_PASSWORD: "abcd1234"
    command: ['postgres', '-c', 'wal_level=logical']
    ports:
       - "5432:5432"
```

```
    networks:
        bubble:
```

> No final deste capítulo veremos informações sobre o comando passado na configuração acima, que configura o `wal_level` para `logical`.

Há também uma carga inicial contendo algumas hospedagens de exemplo. Veja a classe `CargaInicialDoBanco` para mais detalhes.

> Para acessar as alterações presentes no código para este capítulo, rodamos o seguinte comando na raiz do projeto:
>
> `git checkout cap-7`

7.1 Busca de hospedagens

A busca de hospedagens pode ser realizada usando o comando cURL a seguir, ou usando a busca de nome *ms-selecao-hospedagens: listar hospedagens* presente no projeto do Postman:

```
curl -X GET 'http://localhost:8082/hospedagens?nome_destino=SDU'
```

> Tanto no *ms-selecao-hospedagens* quanto nos demais microsserviços já implementados, é possível verificar as siglas de todos os aeroportos do sistema, por meio do enum `Aeroporto` presente no pacote `br.com.alura.msselecaohospedagens.dominio`. No caso, a sigla **SDU** corresponde ao aeroporto Santos Dumont.

Essa busca retorna um resultado semelhante ao seguinte:

```
[
    {
        "cnpj": "42477223000194",
        "nome": "Hostel Amizade",
        "tipo": "HOSTEL",
        "aeroportoMaisProximo": "SDU",
        "diariaAdulto": 202.00,
        "diariaCrianca": 102.00
    },
    {
        "cnpj": "61331527000193",
        "nome": "Hotel San Nicolas",
        "tipo": "HOTEL",
        "aeroportoMaisProximo": "SDU",
        "diariaAdulto": 203.00,
        "diariaCrianca": 103.00
    },
    {
        "cnpj": "72379643000107",
        "nome": "Pousada e Raiz",
        "tipo": "POUSADA",
        "aeroportoMaisProximo": "SDU",
        "diariaAdulto": 201.00,
        "diariaCrianca": 101.00
    }
]
```


A estrutura de uma `Hospedagem` é representada pela classe a seguir:

```
@Entity
@Table(name = "hospedagens")
@NoArgsConstructor(access = AccessLevel.PRIVATE)
@EqualsAndHashCode(of="cnpj")
public class Hospedagem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Getter
```

```
    private String cnpj;

    private String nome;

    @Enumerated(EnumType.STRING)
    private TipoHospedagem tipo;

    @Enumerated(EnumType.STRING)
    private Aeroporto aeroportoMaisProximo;

    @Embedded
    private Diaria diaria;

    public Hospedagem(String cnpj, String nome, TipoHospedagem tipo,
Aeroporto aeroportoMaisProximo, Diaria diaria) {
        this.cnpj = cnpj;
        this.nome = nome;
        this.tipo = tipo;
        this.aeroportoMaisProximo = aeroportoMaisProximo;
        this.diaria = diaria;
    }

    public HospedagemSelecionada selecionar(IdentificadorCompanhiaAerea
idCompanhiaAerea, CpfComprador cpfComprador) {
        return new HospedagemSelecionada(cnpj, nome, diaria,
idCompanhiaAerea, cpfComprador);
    }
}
```

É possível notar que, além dos atributos da classe, do mapeamento ORM e de outras anotações como a do Lombok, há também um método que usaremos no próximo *endpoint*: o `selecionar()`, que criará um objeto do tipo `HospedagemSelecionada`. Esse objeto contém informações-chave da hospedagem, bem como o CPF do viajante que a selecionou e a identificação do voo na companhia aérea (`idCompanhiaAerea`).

```
@EqualsAndHashCode(exclude = "id")
@Entity
```

```
@Table(name = "hospedagens_selecionadas")
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class HospedagemSelecionada {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String cnpj;

    private String nome;

    @Embedded
    private Diaria diaria;

    @Embedded
    private IdentificadorCompanhiaAerea idCompanhiaAerea;

    @Embedded
    private CpfComprador cpf;

    public HospedagemSelecionada(String cnpj, String nome,
            Diaria diaria,
            IdentificadorCompanhiaAerea idCompanhiaAerea, CpfComprador
cpf) {
        this.cnpj = cnpj;
        this.nome = nome;
        this.diaria = diaria;
        this.idCompanhiaAerea = idCompanhiaAerea;
        this.cpf = cpf;
    }
}
```

A identificação do voo na companhia aérea, presente na hospedagem selecionada, será usada pelos microsserviços a serem implementados nos próximos capítulos.

Para selecionar uma hospedagem efetivamente, podemos usar o comando cURL adiante ou a requisição no Postman *ms-selecao-hospedagens: selecionar hospedagem*:

```
curl -X POST 'http://localhost:8082/hospedagens-selecionadas' \
--header 'Content-Type: application/json' \
--data '{
    "cnpj": "42477223000194",
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
    "cpf": "35433320316"
}'
```

Assim que a hospedagem é selecionada, é inserido um registro na tabela `hospedagens_selecionadas`.

A seguir, veremos como o Kafka Connect deve ser configurado para que os registros inseridos nessa tabela sejam também enviados a um tópico.

7.2 Envio de registros do ms-selecao-hospedagens para tópicos do Kafka via Kafka Connect

Para que os registros inseridos na tabela `hospedagens_selecionadas` sejam enviados para o Kafka, podemos seguir os passos já aplicados em capítulos anteriores. Inicialmente, incluímos o *connector plugin* para o Postman no `Dockerfile` que cria a imagem do *cdc-kafka-connect*:

```
RUN echo ">>>>>>>>>> Instalando Plugin do postgres <<<<<<<<<<"
RUN wget  https://repo1.maven.org/maven2/io/debezium/debezium-connector-postgres/2.4.0.Final/debezium-connector-postgres-2.4.0.Final-plugin.tar.gz
RUN tar xvfz debezium-connector-postgres-2.4.0.Final-plugin.tar.gz --directory $KAFKA_CONNECT_PLUGINS_DIR
```

Em seguida, recriamos a imagem:

```
cd docker/cdc-kafka-connect
./dockerizar-kafka-connect.sh
```

Com a imagem recriada, podemos rodar a *infra* da solução novamente:

```
cd docker
./infra-up.sh
```

e subir o *ms-selecao-hospedagens* (usando a IDE de nossa preferência ou via linha de comando):

```
mvn -f ms-selecao-hospedagens/pom.xml spring-boot:start
```

Precisamos então criar o conector entre a tabela no Postgres e o tópico no Kafka. Para isso, podemos criar um arquivo JSON semelhante aos já criados para outros conectores, com algumas diferenças:

- o `name` deve ser único para cada conector. Então, seguindo o padrão de nomenclatura dos conectores anteriores, nomeamos esse conector como `hospedagem-selecionada-conector`;
- por se tratar de uma conexão com o Postgres, o `connector.class` deve ser o `io.debezium.connector.postgresql.PostgresConnector`;
- `plugin.path`: os conectores que acessam o Postgres precisam dessa configuração para definir qual será o plugin a ser usado pelo Debezium para leitura do log do banco. Essa configuração será vista em detalhes no final deste capítulo, quando analisarmos como o Kafka Connect acessa os dados do Postgres.

```
{
    "name": "hospedagem-selecionada-conector",
    "config": {
        "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
        "database.hostname": "postgres",
```

```
        "database.port": "5432",
        "database.user": "postgres",
        "database.password": "abcd1234",
        "database.dbname": "hospedagens",
        "table.include.list": "public.hospedagens_selecionadas",
        "skipped.operations": "u,d,t",
        "topic.prefix": "ms-hospedagens",
        "plugin.name": "pgoutput"
    }
}
```

Para instalarmos esse conector, podemos rodar o comando `./hospedagem-selecionada-conector.sh` presente em `docker/cdc-kafka-connect`.

Ao realizarmos uma nova seleção de hospedagem, como a do início do capítulo, podemos notar que:

- foi inserido um registro na tabela `hospedagens_selecionadas`:

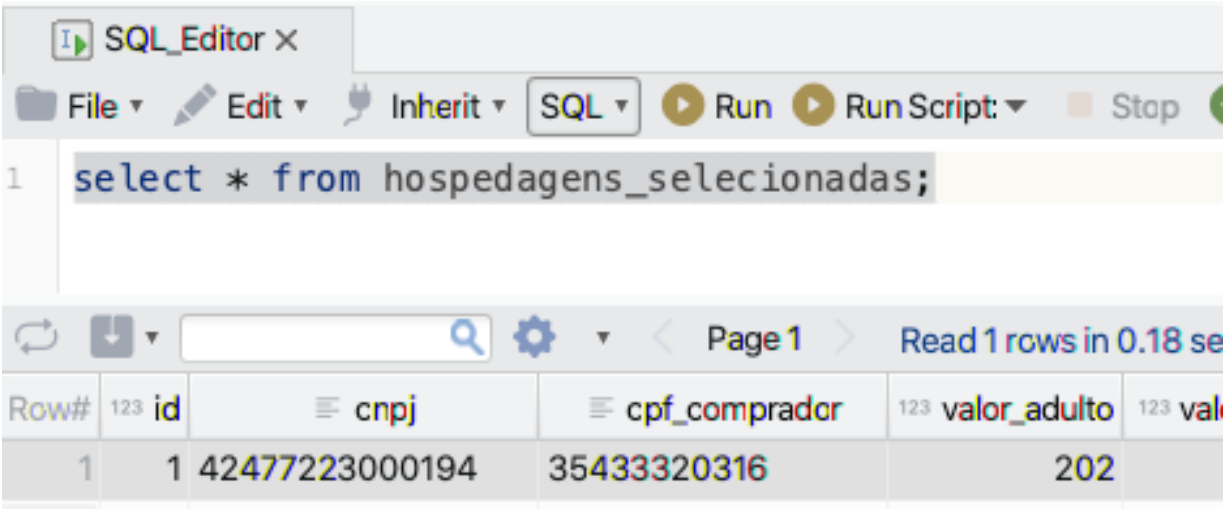


Figura 7.1: Inserção de registro de hospedagem selecionada no Postgres.

- um novo registro foi inserido no tópico `ms-hospedagens.public.hospedagens_selecionadas` (trechos da mensagem omitidos por brevidade):

```
{
   "schema":{
      "type":"struct"
      "fields":[
         0:{...}
         1:{...}
         2:{...}
         3:{...}
         4:{...}
         5:{...}
      ]
      "optional":false
      "name":"ms-hospedagens.public.hospedagens_selecionadas.Envelope"
      "version":1
   }
   "payload":{
      "before":NULL
      "after":{
         "id":1
         "cnpj":"42477223000194"
         "cpf_comprador":"35433320316"
         "valor_adulto":"Tug="
         "valor_crianca":"J9g="
         "identificador":"65136e3571e71b6bcc8b3028"
         "nome":"Hostel Amizade"
      }
      "source":{
         "version":"2.4.0.Final"
         "connector":"postgresql"
         "name":"ms-hospedagens"
         "ts_ms":1698600722066
         "snapshot":"false"
         "db":"hospedagens"
         "sequence":"["26743808","26743912"]"
         "schema":"public"
         "table":"hospedagens_selecionadas"
         "txId":756
         "lsn":26743912
         "xmin":NULL
```

```
      }
      "op":"c"
      "ts_ms":1698600722616
      "transaction":NULL
   }
}
```

É possível notar que, assim como ocorreu na integração com os bancos de dados MongoDB e MySQL, a integração com o Postgres envia para o tópico diversas informações sobre a conexão com o banco de dados, além dos dados do registro na tabela. Podemos, portanto, realizar modificações no conector semelhantes às aplicadas nos conectores com os demais bancos.

Inicialmente, vamos criar um arquivo JSON de atualização do conector, com o conteúdo já presente no JSON de criação:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput"
}
```

Podemos inserir mais algumas configurações, como a de utilização do `JsonConverter`, para que possamos desabilitar o envio do *JsonSchema* nas mensagens:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
```

```
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
}
```

Para rodar essa atualização, podemos rodar o script `./hospedagem-selecionada-conector-atualizar.sh` presente na pasta `docker/cdc-kafka-connect`:

```
curl -X PUT \
  http://localhost:8083/connectors/hospedagem-selecionada-conector/config
\
  -H 'content-type: application/json' \
  -d @hospedagem-selecionada-conector-atualizar.json
```

Após essa alteração no conector, as próximas mensagens presentes no tópico já diminuíram bastante seu tamanho, mas ainda contêm informações além do registro realizado em banco:

```
{
   "before":NULL,
   "after":{
      "id":2,
      "cnpj":"42477223000194"
      "cpf_comprador":"35433320316"
      "valor_adulto":"Tug="
      "valor_crianca":"J9g="
      "identificador":"65136e3571e71b6bcc8b3028"
      "nome":"Hostel Amizade"
   },
   "source":{
      "version":"2.4.0.Final"
      "connector":"postgresql"
      "name":"ms-hospedagens"
```

```
      "ts_ms":1698605486845
      "snapshot":"false"
      "db":"hospedagens"
      "sequence":"["26744264","26938776"]"
      "schema":"public"
      "table":"hospedagens_selecionadas"
      "txId":759
      "lsn":26938776
      "xmin":NULL
   }
   "op":"c"
   "ts_ms":1698605487059
   "transaction":NULL
}
```


Para removermos essas informações, podemos usar o SMT que usamos no MySQL: o `ExtractNewRecordState` . Vamos então, atualizar o JSON novamente:


```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState"
}
```


Agora, após atualização do conector, as novas mensagens possuem apenas o conteúdo da tabela `hospedagens_selecionadas` :

```
{
   "id":3,
   "cnpj":"42477223000194",
   "cpf_comprador":"35433320316",
   "valor_adulto":"Tug=",
   "valor_crianca":"J9g=",
   "identificador":"65136e3571e71b6bcc8b3028",
   "nome":"Hostel Amizade",
}
```


Conforme documentação do Debezium, o SMT `io.debezium.transforms.ExtractNewRecordState` pode ser aplicado em apenas bancos de dados relacionais. O `io.debezium.connector.mongodb.transforms.ExtractNewDocumentState` (usado na integração com o MongoDB), embora tenha o mesmo nome, está em um pacote diferente e é exclusivo para o MongoDB.

Essa documentação pode ser consultada em https://debezium.io/documentation/reference/stable/transformations/event-flattening.html e https://debezium.io/documentation/reference/stable/transformations/mongodb-event-flattening.html#mongodb-new-document-state-extraction

Agora podemos corrigir os valores decimais `valor_adulto` e `valor_crianca`, presentes na mensagem, inserindo a configuração `"decimal.handling.mode"` com o valor `"string"`:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
```

```
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string"
}
```

Com essa alteração, a mensagem passa a conter os valores desses campos em um formato amigável para humanos e microsserviços consumidores:

```
{
   "id":4
   "cnpj":"42477223000194",
   "cpf_comprador":"35433320316",
   "valor_adulto":"202.00",
   "valor_crianca":"102.00",
   "identificador":"65136e3571e71b6bcc8b3028",
   "nome":"Hostel Amizade"
}
```

Precisamos ainda alterar a maneira com que a chave da mensagem é armazenada no tópico. Ela está no seguinte formato:

```
Struct{id=4}
```

Seguindo o que fizemos no capítulo anterior, basta inserirmos o SMT `ExtractField$Key` no JSON de atualização do conector:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
```

```
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"id"
}
```

Com isso, a chave da mensagem passa a ter apenas seu valor.

Após as alterações realizadas neste capítulo, nosso diagrama de solução ficará assim:

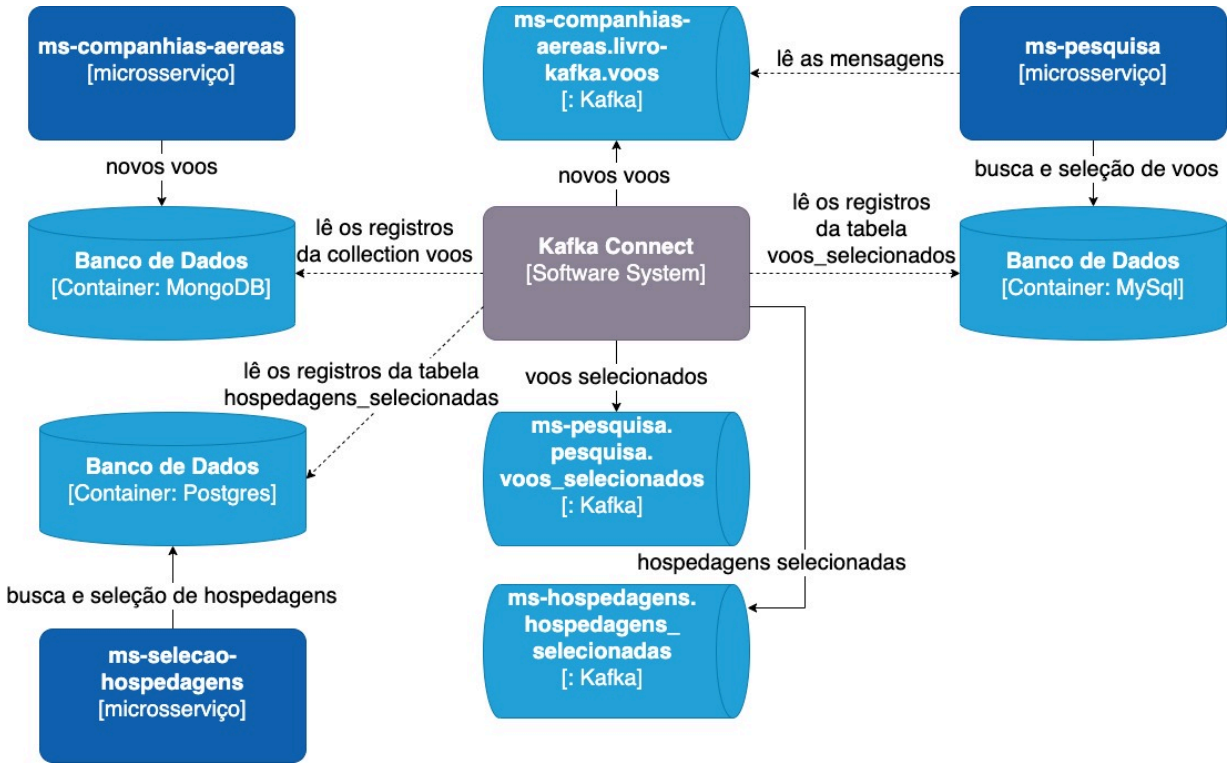


Figura 7.2: Inclusão do ms-selecao-hospedagens na solução.

7.3 Leitura de dados do Postgres pelos conectores CDC

Assim como no MongoDB e no MySQL, é necessário que o Postgres seja devidamente preparado para a leitura dos dados a ser realizada pelo Kafka Connect.

O Postgres trabalha com o conceito de WAL (*Write-ahead Log*), em que basicamente todas as mudanças realizadas no banco de dados são registradas nesse *log*, de forma contínua, antes mesmo de serem persistidas nas tabelas do banco. Esse procedimento visa garantir que, em caso de falha numa transação, possa-se recuperar

o log e refazê-la. Além disso, com ele pode-se inclusive realizar *backups*, recuperações e até reversões do banco de dados.

> Mais informações sobre o WAL em https://www.postgresql.org/docs/current/logicaldecoding-explanation.html

Para garantir a segurança desse *log*, o Postgres oferece mecanismos de replicação "somente leitura" desse *log*. Assim, é possível realizar leituras mais seguras dessas informações, sem possibilidade de alteração delas.

> Mesmo com a possibilidade de realizarmos leituras mais seguras dos dados, é importante reforçar que são informações que podem ser confidenciais, e que mesmo uma leitura desse tipo de dados pode acarretar sérios problemas de segurança. Logo, é importante a criação de usuários exclusivos para o Kafka Connect, controlando seus acessos para possam ler somente o que precisam, e nada mais.

Essa leitura é realizada por meio de decodificação do conteúdo do WAL e, para que essa decodificação seja feita, pode-se usar um dos *plugins* a seguir:

- `pgoutput` : disponível a partir da versão 10 do Postgres. Dispensa instalação prévia, visto já estar disponível no banco. Esse é o *plugin* que utilizamos nos conector construído neste capítulo (descrito na configuração `plugin.path`);
- `decoderbufs` : decodifica o *log* para o formato *Protocol Buffers* e é mantido pela comunidade Debezium, com contribuições do time que mantém o Postgres. Para utilizá-lo, é necessário usar o conector Postgres mantido pela Debezium. É o valor padrão para a configuração `plugin.path` caso o *connector plugin* usado seja o da Debezium. Mais informações sobre ele podem ser

obtidas no link https://github.com/debezium/postgres-decoderbufs;

- `wal2json` : decodifica o *log* para o formato JSON, um formato mais amigável para humanos, mas que torna a decodificação maior e mais lenta para ser lida e processada por computadores. Esse plugin é suportado apenas pelo *connector plugin* para Postgres disponibilizado pela Confluent. Mais informações em https://github.com/eulerto/wal2json/blob/master/README.md.

> *Protocol Buffers* (ou simplesmente *protobuf*) é um formato de dados serializados criado pelo Google para troca de informações entre sistemas. É suportado por diversas linguagens de programação. Mais informações sobre esse protocolo podem ser obtidas em https://protobuf.dev/

A maneira com que todos esses componentes se relacionam pode ser representada pela figura a seguir:

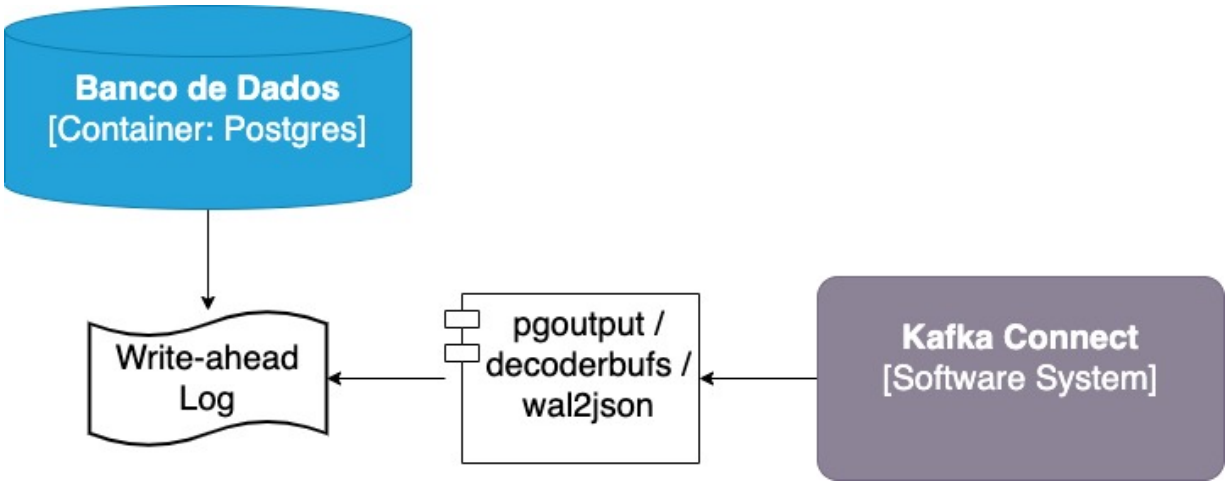


Figura 7.3: Leitura de logs do Postgres realizada pelo Kafka Connect.

Além disso, assim como os demais bancos de dados, é necessário realizar alterações no nível do *log*. No caso do Postgres, é necessário

habilitar a geração para o formato lógico. Essa habilitação pode ser feita de duas formas diferentes:

- alterando a configuração `wal_level` para `logical`, presente no arquivo `postgresql.conf`. Esse procedimento poderia ser feito adicionando um volume no *container* (assim como foi feito com o MongoDB);
- acrescentando o `command` presente no seguinte `docker-compose.yml`: `command: ['postgres', '-c', 'wal_level=logical']`

Esse nível de log registra detalhes ainda mais abrangentes que os demais níveis. Ele captura informações sobre operações de nível mais alto, como operações DML (*Data Manipulation Language*), que são necessárias para replicação lógica ou para capturar mudanças específicas nos dados.

> Essa configuração é suficiente para as versões mais recentes de Postgres. Caso a versão de banco de dados seja mais antiga, pode ser importante consultar a documentação do *connector plugin* utilizado, para verificar se mais alterações precisam ser feitas. No caso do Debezium (usado neste livro) ela pode ser consultada em https://debezium.io/documentation/reference/stable/connectors/postgresql.html

Para consultarmos o *log* do Postgres, podemos usar o comando a seguir. Os logs serão exibidos no formato binário. O `pg_waldump` é o utilitário de leitura disponibilizado na instalação do Postgres; o argumento `-p` indica a pasta onde se encontra os arquivos de log; e o argumento `000000010000000000000001` é o arquivo de *log* a ser lido.

```
pg_waldump -p /var/lib/postgresql/data/pg_wal 000000010000000000000001
```

Como resposta a esse comando, aparecerá o *log* do banco, onde os comandos aplicados ao banco de dados são exibidos no formato de

caracteres. A seguir, apenas um trecho da resposta, para demonstração:

```
rmgr: Heap        len (rec/tot):    111/   111, tx:        745, lsn:
0/019539C8, prev 0/01953998, desc: INSERT off: 8, flags: 0x08, blkref #0:
rel 1663/16384/16389 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:        745, lsn:
0/01953A38, prev 0/019539C8, desc: INSERT_LEAF off: 8, blkref #0: rel
1663/16384/16397 blk 1
rmgr: Transaction len (rec/tot):     46/    46, tx:        745, lsn:
0/01953A78, prev 0/01953A38, desc: COMMIT 2023-11-05 18:23:22.954224 UTC
rmgr: Heap        len (rec/tot):    113/   113, tx:        746, lsn:
0/01953AA8, prev 0/01953A78, desc: INSERT off: 9, flags: 0x08, blkref #0:
rel 1663/16384/16389 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:        746, lsn:
0/01953B20, prev 0/01953AA8, desc: INSERT_LEAF off: 9, blkref #0: rel
1663/16384/16397 blk 1
rmgr: Transaction len (rec/tot):     46/    46, tx:        746, lsn:
0/01953B60, prev 0/01953B20, desc: COMMIT 2023-11-05 18:23:22.977513 UTC
rmgr: Heap        len (rec/tot):    115/   115, tx:        747, lsn:
0/01953B90, prev 0/01953B60, desc: INSERT off: 10, flags: 0x08, blkref #0:
rel 1663/16384/16389 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:        747, lsn:
0/01953C08, prev 0/01953B90, desc: INSERT_LEAF off: 10, blkref #0: rel
1663/16384/16397 blk 1
rmgr: Transaction len (rec/tot):     46/    46, tx:        747, lsn:
0/01953C48, prev 0/01953C08, desc: COMMIT 2023-11-05 18:23:22.994658 UTC
rmgr: Heap        len (rec/tot):    114/   114, tx:        748, lsn:
0/01953C78, prev 0/01953C48, desc: INSERT off: 11, flags: 0x08, blkref #0:
rel 1663/16384/16389 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:        748, lsn:
0/01953CF0, prev 0/01953C78, desc: INSERT_LEAF off: 11, blkref #0: rel
1663/16384/16397 blk 1
rmgr: Transaction len (rec/tot):     46/    46, tx:        748, lsn:
0/01953D30, prev 0/01953CF0, desc: COMMIT 2023-11-05 18:23:23.016194 UTC
rmgr: Heap        len (rec/tot):    116/   116, tx:        749, lsn:
0/01953D60, prev 0/01953D30, desc: INSERT off: 12, flags: 0x08, blkref #0:
rel 1663/16384/16389 blk 0
```

Para a conversão do log obtido pelo comando acima num formato mais legível por humanos, pode-se usar um dos plugins presentes na página https://wiki.postgresql.org/wiki/Logical_Decoding_Plugins

7.4 Resumo

Neste capítulo, vimos como lidar com a busca e seleção de hospedagens para o nosso viajante. Também vimos como integrar o Kafka Connect em uma base de dados Postgres para enviar as hospedagens selecionadas para um tópico Kafka.

No próximo capítulo, veremos mais algumas SMTs que serão muito úteis. Também começaremos a montar o *ms-pacote*, solução que montará os pacotes de viagem com base nas passagens e hospedagens selecionadas pelos viajantes.

Capítulo 8
Boas práticas para nomes de tópicos e outros SMTs interessantes

Já implementamos diversos microsserviços na nossa solução. Também pudemos entender como o Kafka Connect se conecta a diversos bancos de dados. Vamos então realizar alguns ajustes finos na solução antes de partirmos para outros conceitos relacionados ao Kafka.

8.1 Nomeação de tópicos

No capítulo 3, vimos que os conectores do Kafka Connect não possuem uma parametrização que determine diretamente o nome do tópico. Em vez disso, realiza-se uma concatenação de 3 parâmetros diferentes:

- `topic.prefix` : identifica a fonte de dados no Kafka Connect;
- `table.include.list` : identifica a(s) tabela(s) a ser(em) lidas pelo Kafka Connect. Cada tabela corresponderá a um tópico diferente;
- `collection.include.list` : idem ao anterior, mas para MongoDB (onde usamos *collections* e não tabelas pra persistência de dados).

Com isso, temos os seguintes tópicos construídos até o momento:

- `ms-companhias-aereas.livro-kafka.voos` (onde armazenamos os novos voos criados no *ms-companhias-aereas*);
- `ms-pesquisa.pesquisa.voos_selecionados` (para onde os voos selecionados pelos viajantes são enviados);

- `ms-hospedagens.public.hospedagens_selecionadas` (feito para persistir as hospedagens selecionadas).

É possível notar que, para nomear os tópicos, foi inserido no campo `topic.prefix` o nome do microsserviço que insere dados no banco lido pelo Kafka Connect. Dessa forma, é possível descrever, por meio do tópico, qual o microsserviço produtor de mensagens para o tópico, bem como a tabela (ou a *collection*) que será lida pelo Kafka Connect.

Entretanto, assim como pode ocorrer com tabelas de banco de dados, as empresas podem possuir seus padrões de nomenclatura de tópicos. E essa regra pode, portanto, ser um problema nesse cenário.

Veremos então como nomear os tópicos de destino das mensagens sem a necessidade de ficarmos presos ao padrão do Kafka Connect.

8.2 Recomendações para nomes de tópicos

O PROBLEMA DA NOMEAÇÃO

Como fornecer bons nomes a métodos, variáveis, classes e tabelas? Essa definitivamente não pode ser uma questão a ser ignorada. Programadores do mundo todo enfrentam esse problema quando estão programando, ou mesmo preparando a infraestrutura de suas aplicações. Ele é tão recorrente que diversos livros o citam e buscam soluções, como Block (2018), Fowler (2018), Martin (2008), McConnell (2004), e Thomas e Hunt (2019). Esses livros não buscaram exatamente fornecer sugestões para a pessoa programadora nomear melhor os tópicos que ela precisa criar. Entretanto, veremos a seguir que alguns autores aplicaram conceitos citados nessas fontes também nesse caso.

De uma forma geral, as referências citadas anteriormente nos ensinam que a clareza deve ser priorizada na nomeação. Mas também citam outras características de uma boa nomeação:

- Os nomes devem revelar o propósito do que está se construindo.
- Dependendo dos times que lidarão com o que foi construído, é possível que nomes diferentes sejam dados para o mesmo conceito. Nesse caso, é importante criar uma linguagem comum a todos os envolvidos. Usar nomes a partir do domínio da solução ou do problema pode ajudar.
- Bons nomes requerem prática. Então devemos praticar.

Há também livros como o escrito por Shapira, Narkhede e Palino (2017) que fornecem regras de sintaxe para os nomes de tópicos. Segundo eles, recomenda-se que esses:

- não devem começar com dois *underscores*, visto que tópicos nomeados dessa forma são considerados como sendo internos ao *cluster*;
- não devem conter *underscores* e pontos em nomes de tópicos presentes em um mesmo *cluster*, mas um ou outro. Essa regra diz respeito à coleta de métricas por tópico realizada pelo Kafka, na qual essas são renomeadas com *underscores* no lugar dos pontos. Dessa forma, evita-se confusões na associação entre tópicos e suas respectivas métricas.

Repare que essas regras são importantes para o bom funcionamento do tópico em determinadas situações. Mas nenhuma delas diz respeito à clareza do nome ou a outros requisitos.

É possível verificar que dois dos tópicos que criamos não estão seguindo as recomendações mínimas citadas anteriormente, visto que ambos possuem pontos e *underscores* no mesmo tópico:

- `ms-pesquisa.pesquisa.voos_selecionados`
- `ms-hospedagens.public.hospedagens_selecionadas`

Além disso, o Robin Moffatt, *Principal Developer Advocate* da Confluent, em uma conversa no fórum da Confluent (https://forum.confluent.io/t/how-to-paint-a-bike-shed-kafka-topic-naming-conventions/280) fornece alguns *links* interessantes, como o https://www.kadeck.com/blog/kafka-topic-naming-conventions-5-recommendations-with-examples, com as seguintes recomendações:

- Em termos de regras de nomenclatura, não faça pouco, mas também não exagere! É importante que todo o time de

engenharia conheça e concorde com as regras criadas.

- Favoreça os nomes com letras minúsculas e com pontos ou *underscores*, em detrimento de letras maiúsculas e traços.
- Para evitar problemas relacionados a nomes diferentes para os mesmos dados (aqueles citados no início do capítulo) pode-se utilizar um critério baseado em quem produz os dados. Ou seja: quem é o produtor? E usar esse critério como um possível desempate.
- Pensar em domínio e em subdomínios, até chegar nos dados desejados. Exemplo:
  - `lojas.vendas.livros.informatica` : nesse caso, estamos armazenando as vendas de livros de informática das lojas. Essas lojas podem ser um conjunto classificado por algum motivo (uma rede de lojas de um único dono, por exemplo).
  - `transacoes.venda.cartoes.credito` : as transações de venda cujo meio de pagamento foram cartões de crédito.
- Como não é possível classificar os tópicos usando qualquer mecanismo nativo do Kafka (como grupos ou pastas) recomenda-se utilizar os nomes dos tópicos para esse fim, fazendo com que tópicos com dados semelhantes possuam o início do nome também semelhante. As ferramentas de visualização de tópicos, inclusive, possuem recursos para busca por nome que permitem encontrá-los mais facilmente dessa forma.
- Os nomes não devem conter:
  - nomes de aplicações, sejam elas produtoras ou consumidoras (podemos dizer que todos os tópicos criados para o Alura Viagens infringem essa regra). Dessa forma, podemos futuramente alterar a aplicação produtora de mensagens, se necessário, ou mesmo incluir novos consumidores.
  - números correspondentes a versões. Essa regra pode ser boa para *endpoints* REST, mas não para nomeação de tópicos. Assim, evitamos problemas relacionados a limites de criação de tópicos ou partições.

Pode ser razoável resumirmos essas recomendações em duas:

- seguir as convenções técnicas;
- favorecer nomes baseados em conceitos do domínio da solução, e não na tecnologia utilizada.

> Por último, é importante lembrar que o Kafka não permite a renomeação de tópicos. Então, caso seja necessário trocar o nome de um tópico, será necessária a criação de um outro tópico e, em seguida, mover todas as mensagens do tópico antigo para o novo.

8.3 Renomeação de tópicos no Alura Viagens com o RegexRouter

Para a renomeação dos tópicos utilizados no Alura Viagens, o time de arquitetura determinou algumas regras:

- os nomes serão todos em letras minúsculas, com uso de ponto para possível classificação;
- os nomes seguirão os conceitos do domínio da solução, dividindo-o em subdomínios se necessário.

A Apache oferece um SMT para que a renomeação do tópico de destino seja possível: o `org.apache.kafka.connect.transforms.RegexRouter`. Ele, basicamente, recebe os seguintes parâmetros:

- **uma expressão regular**, que informa quais nomes devem ser lidos para serem substituídos. Esse parâmetro deve ser escrito no formato `transforms.<nome-do-transformer>.regex`, onde `<nome-do-transformer>` é o nome declarado no parâmetro `transforms` do conector;

- **o nome do tópico** que deve receber os dados coletados do conector. E, seguindo o formato acima, esse parâmetro deve ser escrito no formato `transforms.<nome-do-transformer>.replacement`.

> Devido à configuração `table.include.list` (ou `collection.include.list` em conexões com o MongoDB) podemos ter mais de um tópico de destino. Logo, pode ser necessário selecionar qual nome deve ser renomeado. É por isso que temos a primeira configuração acima.

Vamos ver como configurar esse SMT em cada um dos conectores a seguir.

Renomeação do tópico ms-companhias-aereas.livro-kafka.voos

Para determinarmos o novo nome para o tópico `ms-companhias-aereas.livro-kafka.voos`, vamos usar o seguinte raciocínio:

- o tópico recebe voos;
- mas não são quaisquer voos. São novos voos, disponibilizados pelas companhias aéreas para que possam ser vendidos pelo Alura Viagens;
- podemos ter voos de outros tipos (ou outros estados para a abstração `Voo`) como voos selecionados. E podemos ignorar sua origem. Assim, não precisamos fazer referências às companhias aéreas no nome do tópico.

Logo, podemos pensar em um nome como `voos.novos`, ou mesmo `voos.disponiveis`. Dessa forma, atendemos à regra de domínio e depois subdomínio. No caso, os sufixos `novos` e `disponiveis` são possíveis estados de um voo.

Mas os voos que podem ser selecionados também estão disponíveis. Só não são mais novos. Então usaremos o nome `voos.novos` mesmo. Vamos atualizar o `docker-compose.yml`, fazendo com que ele deixe de

criar o tópico `ms-companhias-aereas.livro-kafka.voos` e passe a criar o tópico novo:

```
kafka-topics --create --bootstrap-server kafka:29092 --partitions 1 --
replication-factor 1 --topic voos.novos
```

Para aplicarmos a SMT `org.apache.kafka.connect.transforms.RegexRouter` no arquivo `novos-voos-conector-atualizar.json`, vamos seguir os seguintes passos:

- nomear essa nova SMT de `rename`
- declará-la na configuração `transforms`, fazendo com que ela fique no formato `"transforms": "rename,extractNewDocumentState",`
- inserir os demais parâmetros no conector:
  - `"transforms.rename.type": "org.apache.kafka.connect.transforms.RegexRouter",`
  - `"transforms.rename.regex": ".*",`
  - `"transforms.rename.replacement": "voos.novos",`

Ao inserir esses novos parâmetros no arquivo `novos-voos-conector-atualizar.json`, ele ficará desta forma:

```
{
   "connector.class": "io.debezium.connector.mongodb.MongoDbConnector",
   "mongodb.connection.string": "mongodb://mongodb:27017",
   "mongodb.user": "user",
   "mongodb.password": "password",
   "database.include.list": "livro-kafka",
   "collection.include.list": "livro-kafka.voos",
   "key.converter": "org.apache.kafka.connect.json.JsonConverter",
   "key.converter.schemas.enable": "false",
   "skipped.operations": "u,t",
   "topic.prefix": "ms-companhias-aereas",
   "value.converter": "org.apache.kafka.connect.json.JsonConverter",
   "value.converter.schemas.enable": "false",

   "transforms": "extractNewDocumentState,rename",

   "transforms.rename.type":
```

```
"org.apache.kafka.connect.transforms.RegexRouter",
   "transforms.rename.regex": ".*",
   "transforms.rename.replacement": "voos.novos",

"transforms.extractNewDocumentState.type":"io.debezium.connector.mongodb.transforms.ExtractNewDocumentState",
   "transforms.extractNewDocumentState.drop.tombstones": "false",
}
```

Foram inseridos pulos de linha no arquivo apenas para facilitar a visualização. Eles não afetam o processamento da atualização. Após as alterações acima, podemos rodar o `./infra-up.sh` novamente. Assim:

- o novo tópico será criado
- o conector será atualizado

Para removermos o tópico antigo, temos 3 alternativas:

- rodar o comando `docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --delete --topic ms-companhias-aereas.livro-kafka.voos`
- via Redpanda, clicar no ícone de lata de lixo que aparece quando passamos o mouse no canto direito da lista de tópicos, conforme figura a seguir:

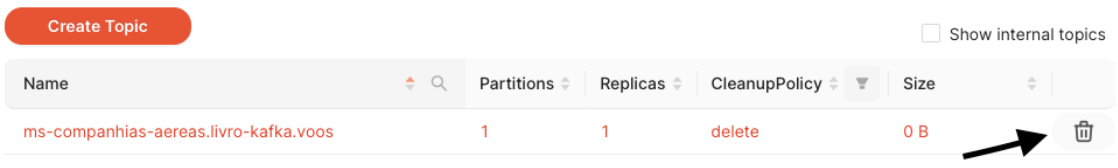


Figura 8.1: Remoção do tópico via Redpanda.

- apenas reiniciar a infraestrutura da aplicação, rodando `./infra-down.sh && ./infra-up.sh`

Após essas alterações, a lista de tópicos existentes no Kafka deve ser a seguinte:

- `ms-hospedagens.public.hospedagens_selecionadas`

- `ms-pesquisa-schemahistory`
- `ms-pesquisa.pesquisa.voos_selecionados`
- `voos.novos`

Também precisamos alterar o tópico que será lido pelo *ms-pesquisa*. Vamos ao seu arquivo `application.yml` e, na configuração `topico.voos.nome`, a deixamos assim:

```
topico.voos.nome: voos.novos
```

Para testarmos se as alterações funcionaram, vamos subir o *ms-companhias-aereas* e o *ms-pesquisa*:

```
mvn -f ms-companhias-aereas/pom.xml compile spring-boot:start
mvn -f ms-pesquisa/pom.xml compile spring-boot:start
```

E, logo em seguida, podemos cadastrar um novo voo:

```
curl --location 'http://localhost:8080/voos' \
--header 'Content-Type: application/json' \
--data '{
   "origem":"CGH",
   "partida":"2024-08-14T18:38:39.072390",
   "destino":"SDU",
   "chegada":"2024-08-14T20:38:39.072451",
   "companhiaAerea":"GOL",
   "precoAdulto": 1000.00,
   "precoCrianca": 500.00
}'
```

Assim que fizermos isso, é possível verificar que a mensagem foi enviada para o tópico recém-criado:

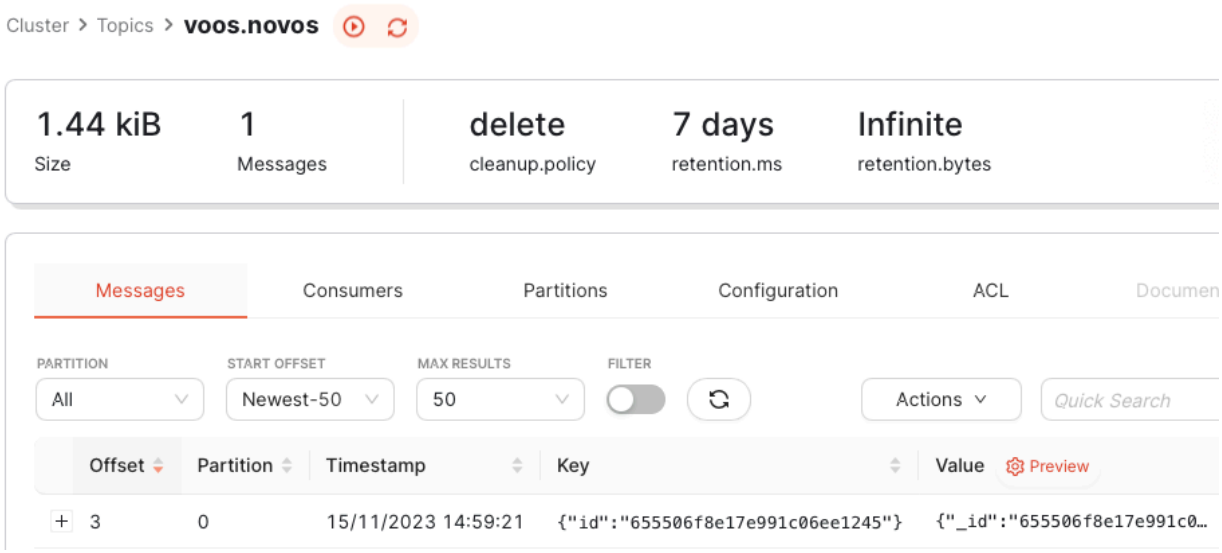


Figura 8.2: Mensagem presente no novo tópico.

Também é possível ver que a base de dados do *ms-pesquisa* contém o novo voo:

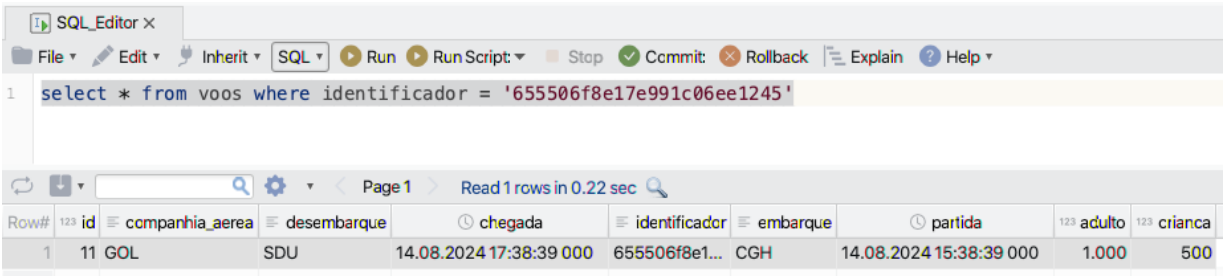


Figura 8.3: Novo voo disponível para compra.

Renomeação do tópico ms-pesquisa.pesquisa.voos_selecionados

Para efetuarmos o mesmo procedimento no tópico relacionado ao *ms-pesquisa*, vamos seguir os mesmos passos realizados na renomeação do tópico anterior.

Em relação ao nome do tópico, seguindo a ideia de que:

- estamos lidando com voos;
- e que dessa vez os voos foram selecionados;

podemos pensar em algo como `voos.selecionados`.

Precisamos então:

- alterar o `docker-compose.yml` para deixar de criar o tópico `ms-pesquisa.pesquisa.voos_selecionados` e criar o novo tópico:

```
kafka-topics --create --bootstrap-server kafka:29092 --partitions 1 --replication-factor 1 --topic voos.selecionados
```

- inserir os parâmetros correspondentes ao SMT `org.apache.kafka.connect.transforms.RegexRouter` no arquivo `novos-voos-conector-atualizar.json`:

```
"transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
"transforms.rename.regex": ".*",
"transforms.rename.replacement": "voos.novos",
```

- inserir a nova SMT na lista de transformadores do conector:

```
"transforms": "extractNewDocumentState,extractKey,rename",
```

O arquivo, após as alterações, ficará desta maneira:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
```

```
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"id",
    "decimal.handling.mode": "string",

    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "voos.selecionados"
}
```

Vamos também reiniciar a infraestrutura para que o novo tópico seja criado, bem como para que o conector seja atualizado:

```
./infra-down.sh && ./infra-up.sh
```

Para testarmos as alterações, o *ms-pesquisa* precisa estar no ar:

```
mvn -f ms-pesquisa/pom.xml compile spring-boot:start
```

Podemos então selecionar um voo específico:

```
curl -X POST 'http://localhost:8081/voos-selecionados' \
--header 'Content-Type: application/json' \
--data '{
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3027",
    "adultos": 1,
    "criancas": 2,
    "cpfComprador": "75625404704"
}'
```

Esse voo selecionado deve estar presente no tópico `voos.selecionados`:

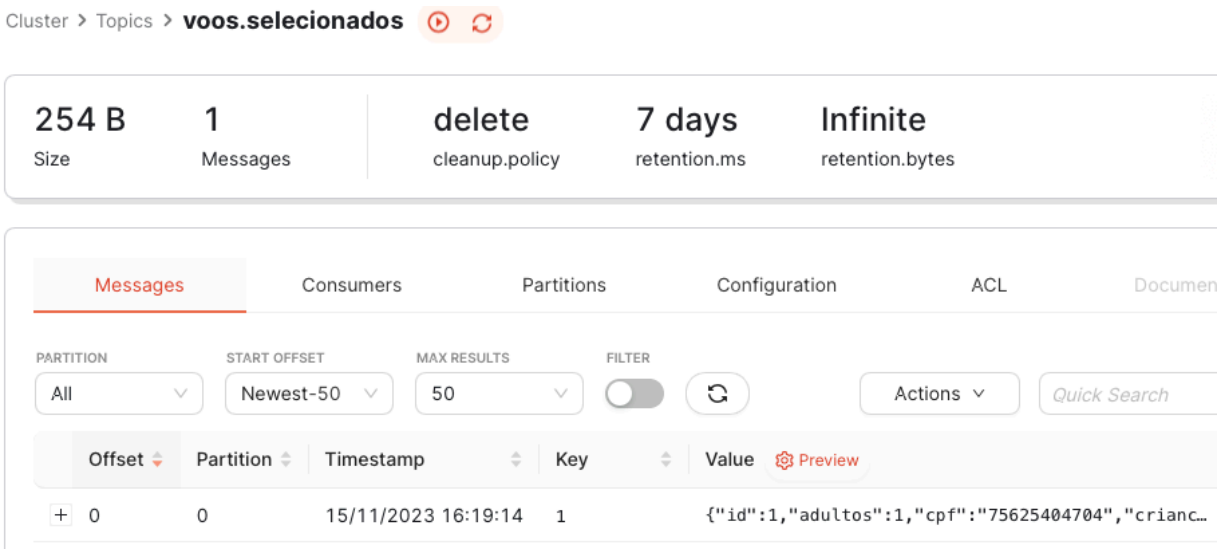


Figura 8.4: Novo voo disponível para compra.

Renomeação do tópico ms-hospedagens.public.hospedagens_selecionadas

De forma resumida, vamos seguir os passos anteriores:

- definição de novo nome, de acordo com as regras que definimos anteriormente. Neste caso, estamos lidando com o conceito de hospedagens, que são selecionadas pelos viajantes. Então, podemos renomear o tópico para `hospedagens.selecionadas`;
- alterar o `docker-compose.yml` para que ele crie o novo tópico, em detrimento do antigo:

```
kafka-topics --create --bootstrap-server kafka:29092 --partitions 1 --
replication-factor 1 --topic hospedagens.selecionadas
```

- inclusão da SMT no arquivo `hospedagem-selecionada-conector-atualizar.json`:

```
"transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
"transforms.rename.regex": ".*",
"transforms.rename.replacement": "hospedagens.selecionadas"
```

- inclusão da nova SMT na lista de SMTs do conector:

```
"transforms": "extractNewDocumentState,extractKey,rename",
```

Essas modificações fazem com que o `hospedagem-selecionada-conector-atualizar.json` fique desta forma:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"id",
    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "hospedagens.selecionadas"
}
```

Para testarmos as alterações realizadas:

- reiniciamos a infra:

```
./infra-down.sh && ./infra-up.sh
```

- subimos o *ms-selecao-hospedagens*:

```
mvn -f ms-selecao-hospedagens/pom.xml spring-boot:start
```

- selecionamos uma hospedagem:

```
curl -X POST 'http://localhost:8082/hospedagens-selecionadas' \
--header 'Content-Type: application/json' \
--data '{
    "cnpj": "42477223000194",
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
    "cpf": "35433320316"
}'
```

- e verificamos o tópico `hospedagens.selecionadas`, em busca da mensagem enviada, com base na hospedagem selecionada:

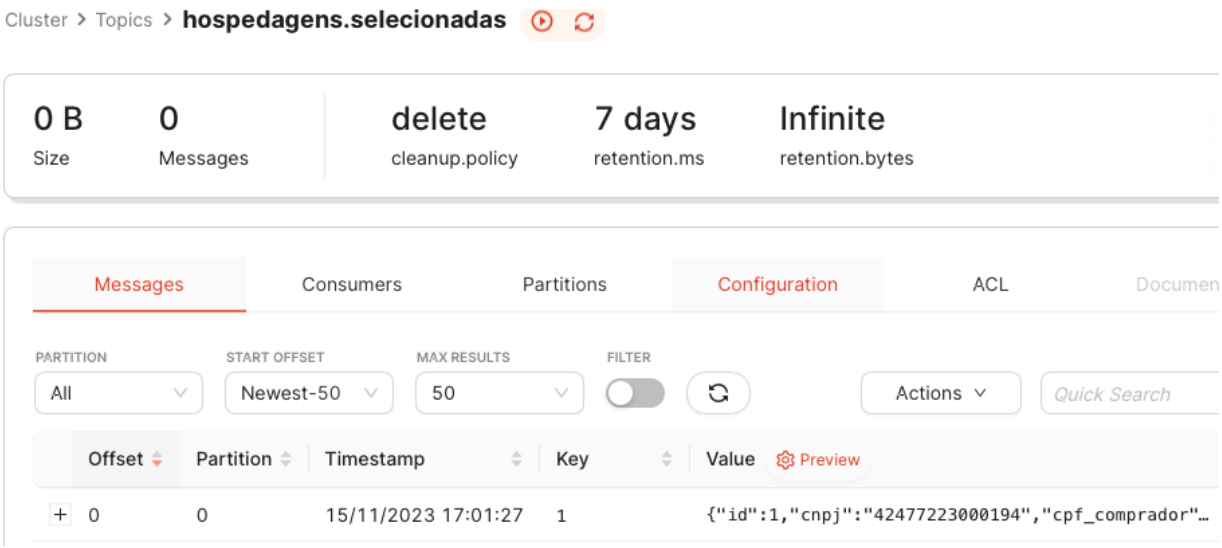


Figura 8.5: Hospedagem selecionada, presente no novo tópico.

Após esses ajustes, a lista de tópicos presentes no Kafka deve ser a seguinte:

- `hospedagens.selecionadas`
- `ms-pesquisa-schemahistory`
- `voos.novos`
- `voos.selecionados`

O tópico `ms-pesquisa-schemahistory` continuará com esse nome. Mas, caso seja necessário utilizar um tópico com outro nome, basta alterar o parâmetro `schema.history.internal.kafka.topic`, presente no arquivo `voo-selecionado-conector-atualizar.json`. Não há, portanto, necessidade de se usar uma SMT nesse caso para essa alteração.

8.4 Uma SMT necessária: filtros

A equipe de produto da Alura Viagens e Turismo, depois de diversas reuniões, decidiu que inicialmente as viagens que ela comercializará no seu site deverão seguir algumas restrições:

- os preços dos voos não devem ultrapassar o valor de R$2000,00;
- os voos devem partir somente de aeroportos sediados no estado de São Paulo.

Dessa forma, diminui-se consideravelmente a quantidade de voos que serão comercializados no site. A ideia é que a solução entre em produção como um piloto, com apenas alguns voos sendo comercializados, diminuindo-se os riscos de se trabalhar com uma operação maior, com mais voos.

Para atender a esses requisitos, podemos trabalhar com algumas alternativas:

- realizar alterações no *ms-companhias-aereas* de forma a impedir que voos com essas características sejam cadastrados. Embora essa alternativa seja de implementação simples, isso pode tornar o sistema complexo. É importante pensar que

alterações como essas podem ser solicitadas pelo time de produto diversas vezes ao longo do ciclo de vida do *software*. Logo, podemos verificar se existem soluções melhores que essa.
- realizar alterações no *ms-pesquisa*, de forma a impedir que voos com essas características sejam lidos do tópico `novos.voos`, ou mesmo lidos mas não persistidos. Essa solução possui os mesmos problemas relatados na solução anterior.
- solicitar aos colaboradores das companhias aéreas que não cadastrem voos com as características acima. Essa também não é uma boa solução, pois seria um processo altamente sujeito a erros, visto que seria feito por humanos.

Felizmente, existem SMTs que se comportam como filtros de mensagens, selecionando aquelas que devem ir ou não para o tópico Kafka. Podemos então usar esses filtros no conector que envia as mensagens para o tópico `voos.novos`. Assim deixamos essa responsabilidade para o Kafka Connect e não precisamos escrever uma linha sequer de código Java.

As regras de novos voos dizem respeito ao aeroporto de decolagem e ao valor das passagens. Vamos então criar dois filtros:

- um que lide com os valores das passagens;
- outro que lide com os aeroportos de origem.

Vamos chamar o primeiro filtro de `precosAbaixoDe2000Reais` e o segundo, de `origemSomenteDeSP`. Como já vimos, os nomes dos filtros devem ser colocados no parâmetro `transforms` juntamente a outras SMTs existentes.

Precisamos então realizar alterações no arquivo `novos-voos-conector-atualizar.json` para adicionar as novas SMTs, bem como em outros arquivos. Veremos a seguir como fazê-las.

A SMT io.debezium.transforms.Filter

Essa SMT realiza uma leitura de campos determinados da mensagem para filtrá-las. Utiliza também de uma implementação da

especificação JSR-223 para determinar quais campos precisam ser considerados na filtragem.

> A partir da especificação JSR-223 tornou-se possível acessarmos, de nosso código Java, scripts escritos em linguagens como JavaScript, Ruby, Python ou mesmo Groovy. O contrário também é verdadeiro: é possível acessarmos nossos objetos escritos em Java usando essas linguagens.
>
> Maiores informações podem ser lidas em https://www.jcp.org/en/jsr/detail?id=223 , https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html e https://download.oracle.com/otn-pub/jcp/sjp-1_1-pr2-spec-oth-JSpec/sjp-1_1-pr2-spec.pdf?AuthParam=1700422338_a1e07d62fea3baa3404932d7018d54da

O Debezium possui 2 alternativas de linguagens para utilizarmos como linguagem de script:

- Groovy
- JavaScript

Usaremos o Groovy como linguagem de programação, por se tratar de uma linguagem com sintaxe mais próxima ao Java.

Para usarmos essa SMT, devemos, portanto:

- instalar as bibliotecas de SMT (esse passo já foi executado anteriormente, então não faremos novamente);
- instalar as bibliotecas de scripting do Debezium;
- instalar a implementação Groovy da JSR-223.

> Mais detalhes das instalações citadas acima encontram-se em https://debezium.io/documentation/reference/stable/transformations/filtering.html

Para realizar essas instalações, vamos incluir as seguintes linhas no arquivo `Dockerfile` :

```
RUN echo ">>>>>>>>>> Instalando uma implementação da JSR-223 <<<<<<<<<<"
RUN wget https://repo1.maven.org/maven2/org/apache/groovy/groovy-jsr223/4.0.0/groovy-jsr223-4.0.0.jar
RUN mv groovy-jsr223-4.0.0.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-connector-mongodb

RUN wget https://repo1.maven.org/maven2/org/apache/groovy/groovy/4.0.0/groovy-4.0.0.jar
RUN mv groovy-4.0.0.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-connector-mongodb

RUN wget https://repo1.maven.org/maven2/org/apache/groovy/groovy-json/4.0.0/groovy-json-4.0.0.jar
RUN mv groovy-json-4.0.0.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-connector-mongodb
```

> Note que as bibliotecas foram enviadas para a pasta onde se encontra o *connector plugin* do MongoDB. Dessa forma, ele conseguirá usá-las. Se outros *connector plugins* precisassem dessa SMT, as bibliotecas teriam que ser copiadas para eles também.

Também precisamos instalar as seguintes bibliotecas na pasta `debezium-scripting` (que é a pasta onde se encontram instaladas as demais SMTs):

- debezium-core-2.4.0.Final.jar
- debezium-ddl-parser-2.4.0.Final.jar
- debezium-api-2.4.0.Final.jar
- antlr4-runtime-4.10.1.jar

Para isso, inserimos as seguintes linhas no `Dockerfile` de criação da nossa imagem de Kafka Connect:

```
RUN echo ">>>>>>>>>> Instalando bibliotecas restantes do Debezium
<<<<<<<<<<"
RUN wget https://repo1.maven.org/maven2/io/debezium/debezium-
core/2.4.0.Final/debezium-core-2.4.0.Final.jar
RUN mv debezium-core-2.4.0.Final.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-
scripting

RUN wget https://repo1.maven.org/maven2/io/debezium/debezium-ddl-
parser/2.4.0.Final/debezium-ddl-parser-2.4.0.Final.jar
RUN mv debezium-ddl-parser-2.4.0.Final.jar
$KAFKA_CONNECT_PLUGINS_DIR/debezium-scripting

RUN wget https://repo1.maven.org/maven2/org/antlr/antlr4-
runtime/4.10.1/antlr4-runtime-4.10.1.jar
RUN mv antlr4-runtime-4.10.1.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-
scripting

RUN wget https://repo1.maven.org/maven2/io/debezium/debezium-
api/2.4.0.Final/debezium-api-2.4.0.Final.jar
RUN mv debezium-api-2.4.0.Final.jar $KAFKA_CONNECT_PLUGINS_DIR/debezium-
scripting
```

Com o `Dockerfile` pronto, podemos criar nosso primeiro filtro. Para criá-lo, vamos inicialmente declará-lo no parâmetro `transforms` :

```
"transforms": "extractNewDocumentState,rename,precosAbaixoDe2000Reais",
```

E, em seguida, vamos inserir o filtro no arquivo. São 3 parâmetros:

- `transforms.precosAbaixoDe2000Reais.type` : no caso, o valor deve ser `io.debezium.transforms.Filter` ;
- `transforms.precosAbaixoDe2000Reais.language` : a linguagem de script que usaremos para escrever a expressão de filtragem. No caso, devemos inserir `jsr223.groovy` ;
- `transforms.precosAbaixoDe2000Reais.condition` : a expressão, de fato.

Para determinarmos a expressão a ser colocada neste último parâmetro, basta usarmos como regras:

- toda mensagem possui como elemento inicial o `value` (valor);
- para chegarmos até o campo que queremos analisar, precisamos percorrer a árvore do objeto até o campo desejado.

Vamos analisar uma mensagem presente no tópico `voos.novos`:

```
{
   "_id":"655a712792cda252fa757e56"
   "origem":{
      "aeroporto":"CGH"
      "partida":1723671519000
   }
   "destino": {
      "aeroporto":"SDU"
      "chegada":1723678719000
   }
   "companhiaAerea":"GOL"
   "preco":{
      "adulto":"1000.00"
      "crianca":"500.00"
   }
   "_class":"br.com.alura.mscompanhiasaereas.dominio.Voo"
}
```

Para chegarmos aos atributos `preco.adulto` e `preco.crianca`, precisamos percorrer a árvore a partir do elemento implícito `value`. Assim, escrevemos:

```
value.preco.adulto
value.preco.crianca
```

O filtro devidamente escrito no arquivo `novos-voos-conector-atualizar.json` ficará como o seguinte:

```
{
   "connector.class": "io.debezium.connector.mongodb.MongoDbConnector",
   "mongodb.connection.string": "mongodb://mongodb:27017",
   "mongodb.user": "user",
```

```
   "mongodb.password": "password",
   "database.include.list": "livro-kafka",
   "collection.include.list": "livro-kafka.voos",
   "key.converter": "org.apache.kafka.connect.json.JsonConverter",
   "key.converter.schemas.enable": "false",
   "skipped.operations": "u,t",
   "topic.prefix": "ms-companhias-aereas",
   "value.converter": "org.apache.kafka.connect.json.JsonConverter",
   "value.converter.schemas.enable": "false",

   "transforms": "extractNewDocumentState,rename,precosAbaixoDe2000Reais",


"transforms.extractNewDocumentState.type":"io.debezium.connector.mongodb.t
ransforms.ExtractNewDocumentState",
   "transforms.extractNewDocumentState.drop.tombstones": "false",

   "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
   "transforms.rename.regex": ".*",
   "transforms.rename.replacement": "voos.novos",

   "transforms.precosAbaixoDe2000Reais.type":
"io.debezium.transforms.Filter",
   "transforms.precosAbaixoDe2000Reais.language": "jsr223.groovy",
   "transforms.precosAbaixoDe2000Reais.condition": "new
BigDecimal(value.preco.adulto) < 2000 && new
BigDecimal(value.preco.crianca) < 2000",
}
```

Com essas alterações, podemos:

- reiniciar o Kafka Connect;
- atualizar o conector de novos voos;
- subir o *ms-companhias-aereas* e cadastrar alguns voos, para testarmos o novo filtro.

É possível verificar que, ao cadastrarmos um novo voo com preço para adulto no valor de R$1999,00 e outro com valor para criança

nesse mesmo valor, são postadas 2 mensagens no tópico:

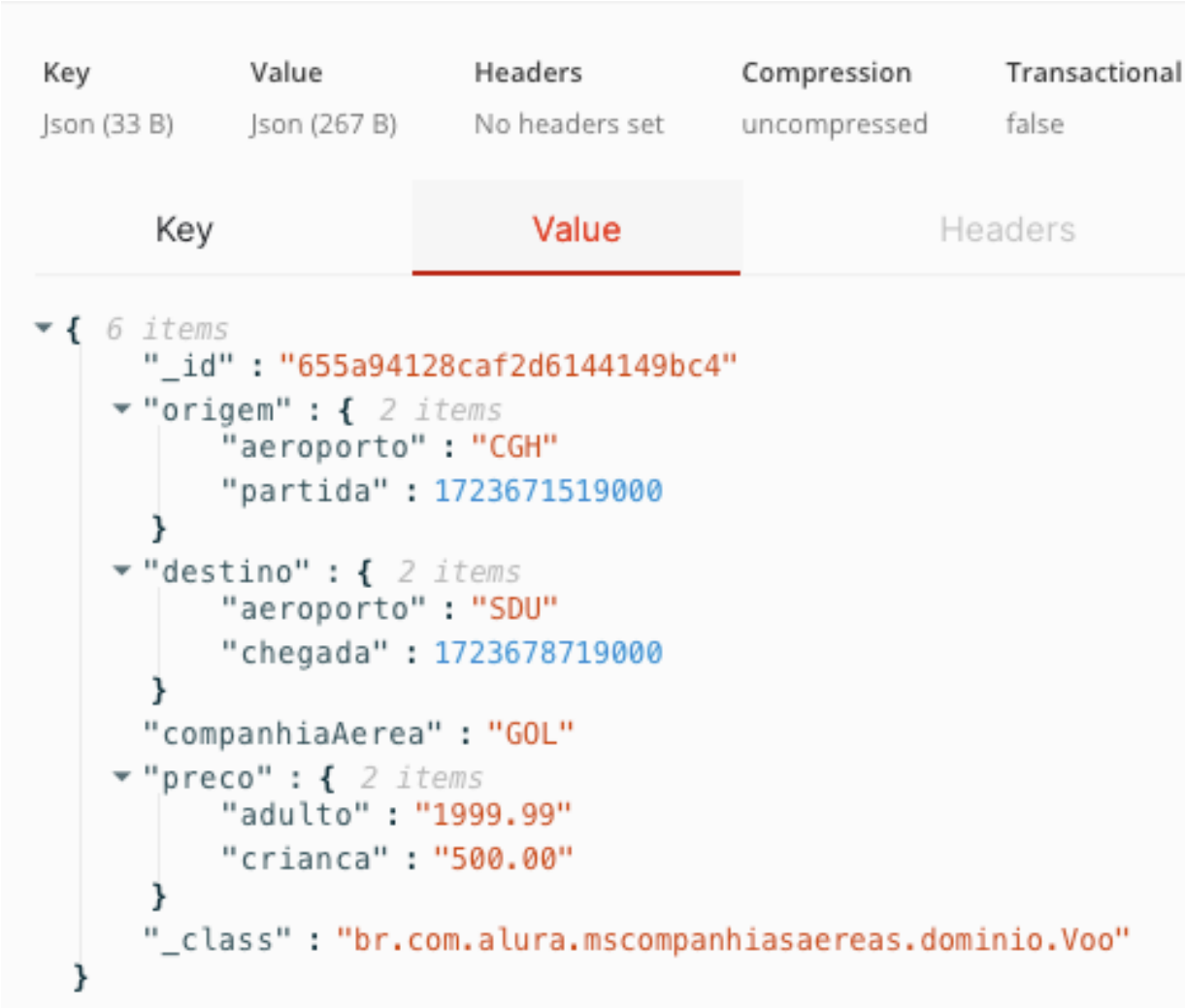


Figura 8.6: Voo com preço máximo para adultos.

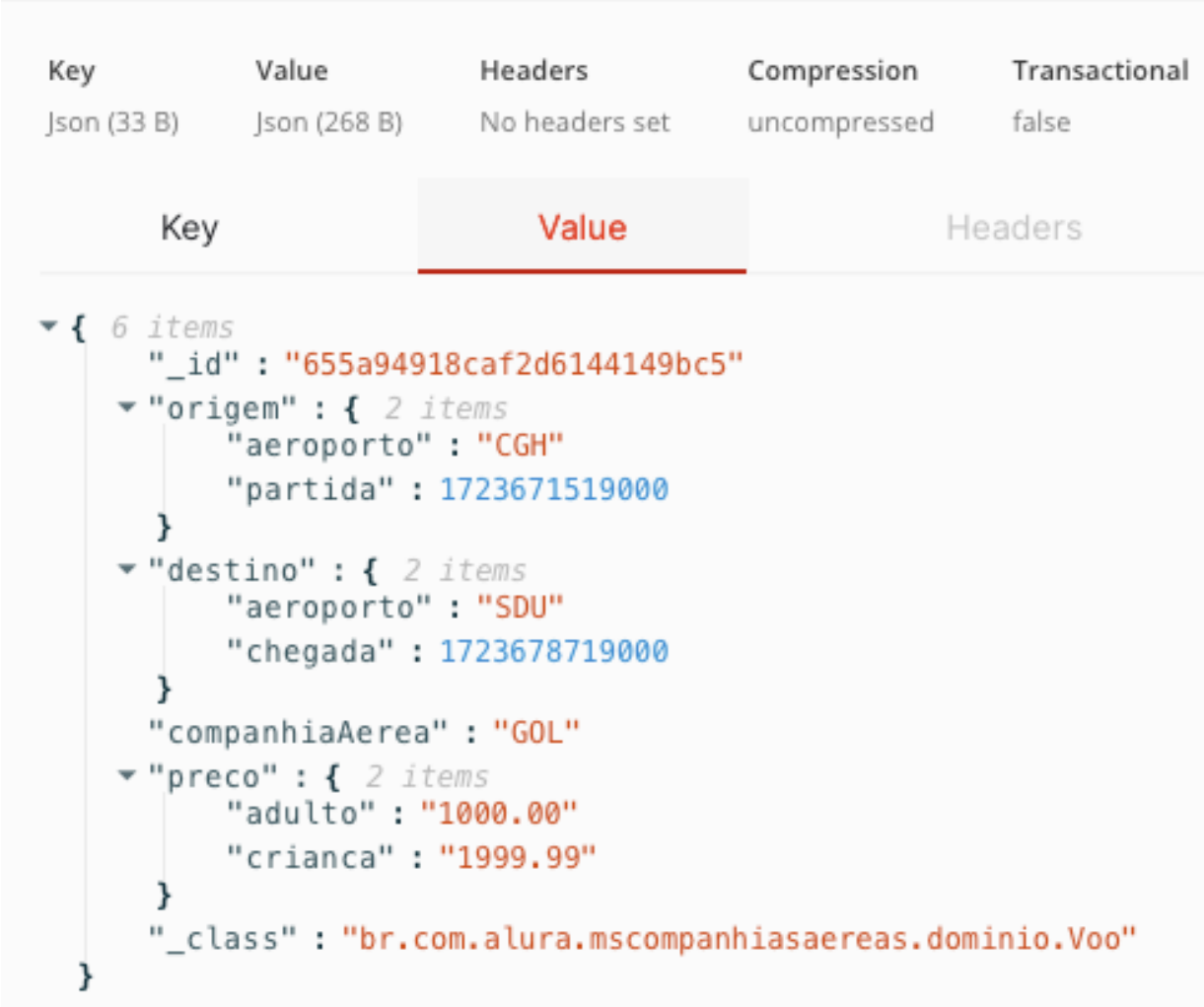


Figura 8.7: Voo com preço máximo para crianças.

Mas se cadastrarmos voos com preços para adultos ou crianças com valores iguais ou superiores a R$2000,00, não serão postadas as mensagens correspondentes a esses voos no tópico, o que comprova que o filtro está funcionando.

> Para mais informações sobre essa SMT, consulte o link: https://debezium.io/documentation/reference/stable/transformations/filtering.html

A SMT io.confluent.connect.transforms.Filter

Além das SMTs do projeto Debezium e da Apache, temos também as disponibilizadas pela Confluent. E veremos como utilizar uma delas que lida com filtragem de mensagens a serem enviadas para um tópico.

Para trabalharmos com as SMTs da Confluent, precisamos instalá-las no nosso Kafka Connect. E temos algumas maneiras de fazer isso:

- acessando o contêiner do Kafka Connect e executando o seguinte comando: `confluent-hub install confluentinc/connect-transforms:latest`;
- assim como fizemos com os conectores da Debezium, instalar as SMTs durante a criação da imagem, via `Dockerfile`.

Usaremos a segunda opção, visto que a primeira funcionaria apenas enquanto o contêiner não for destruído. Além disso, seria necessário apontar para o Kafka Connect o local onde essas SMTs foram instaladas (ou, melhor dizendo, atualizar a variável `plugin.path`).

Para isso, vamos usar o utilitário `confluent-hub`, disponível na imagem de Kafka Connect da Confluent em que nos baseamos para construir a nossa. Devemos incluir as seguintes linhas no `Dockerfile` de criação do Kafka Connect:

```
RUN echo ">>>>>>>>>> Instalando SMTs da Confluent <<<<<<<<<"
RUN confluent-hub install --no-prompt --component-dir
$KAFKA_CONNECT_PLUGINS_DIR confluentinc/connect-transforms:latest
```

Esse comando baixa para a imagem as bibliotecas que contêm as SMTs, bem como algumas documentações e outros arquivos. Também informamos que a instalação deve ser feita na mesma pasta onde estão os *connector plugins* e outros arquivos já instalados anteriormente. Assim, não precisaremos alterar a variável `plugin.path`.

Após a instalação das SMTs, precisamos inserir o filtro no arquivo `novos-voos-conector-atualizar.json`.

Essa SMT possui algumas diferenças em relação à SMT de filtragem do Debezium:

- ela utiliza como sintaxe de filtro o Json path, que é bem similar ao XPath utilizado no passado para arquivos XML, mas para JSON;
- é possível configurar o filtro para incluir ou excluir mensagens que sejam selecionadas pela regra de filtro;
- pode-se filtrar tanto mensagens por valor como também pela chave da mensagem.

> O site https://github.com/json-path/JsonPath contém todas as regras de sintaxe para a escrita de expressões Json path. Recomenda-se sua leitura.

Vamos inicialmente declará-lo no parâmetro `transforms` (considerando que o nome dessa SMT é `origemSomenteDeSP`):

```
"transforms":
"extractNewDocumentState,rename,precosAbaixoDe2000Reais,origemSomenteDeSP"
,
```

E seus parâmetros são:

- `transforms.origemSomenteDeSP.type` : nesse caso, o valor deve ser `io.confluent.connect.transforms.Filter` ;
- `transforms.origemSomenteDeSP.filter.condition` : a regra que determina a aplicação do filtro;
- `transforms.origemSomenteDeSP.filter.type` : caso a mensagem (ou o valor) combinem com a regra acima, determina-se por meio desse parâmetro o que será feito com a mensagem. Ou se envia essa mensagem ao tópico (`include`) ou não (`exclude`);
- `transforms.origemSomenteDeSP.missing.or.null.behavior` : determina-se o que deve ser feito no caso de uma mensagem não conter o valor que é usado para filtragem. Pode-se usar os valores `fail` (nesse caso, a SMT lançará uma exceção), `include` (para aplicar

o filtro) ou `exclude` (para não aplicar o filtro). Usaremos o valor `fail`, visto que os campos são de preenchimento obrigatório.

Talvez o parâmetro mais complexo dos citados acima é o `transforms.origemSomenteDeSP.filter.condition`, já que é necessário o conhecimento de Json path, e de como escrever expressões que retornem valores booleanos.

Diferente da SMT anterior, onde usamos Groovy, essa SMT usa obrigatoriamente JavaScript e, portanto, a raiz do objeto é o `$` (cifrão). Precisamos também percorrer a árvore do objeto até chegarmos no atributo que precisamos. No caso, `origem.aeroporto`:

```
$.origem.aeroporto
```

Mas precisamos escrever uma expressão (ou predicado), que retorne um booleano em uma comparação com valores (no caso, as siglas de aeroportos). Vamos colocar inicialmente um dos dois que precisamos colocar:

```
$.origem[?(@.aeroporto == 'CGH')]
```

O símbolo arroba (`@`) é utilizado para indicar o nó que está sendo processado no predicado. Já o símbolo interrogação (`?`) representa uma expressão de filtro.

Podemos também incluir os símbolos E ou OU, respectivamente `&&` e `||`. A condição ficará assim:

```
$.origem[?(@.aeroporto == 'CGH' || @.aeroporto == 'GRU')]
```

Da maneira como criamos o predicado, inserimos os voos de aeroportos com que queremos trabalhar. Logo, o parâmetro `transforms.origemSomenteDeSP.filter.type` deve ter o valor `include`. Ou seja, *inclua os voos dessas origens no tópico de destino*.

Ao escrevermos o filtro no arquivo de atualização do conector `novos-voos-conector`, ele ficará da seguinte forma:

```
{
   "connector.class": "io.debezium.connector.mongodb.MongoDbConnector",
   "mongodb.connection.string": "mongodb://mongodb:27017",
   "mongodb.user": "user",
   "mongodb.password": "password",
   "database.include.list": "livro-kafka",
   "collection.include.list": "livro-kafka.voos",
   "key.converter": "org.apache.kafka.connect.json.JsonConverter",
   "key.converter.schemas.enable": "false",
   "skipped.operations": "u,t",
   "topic.prefix": "ms-companhias-aereas",
   "value.converter": "org.apache.kafka.connect.json.JsonConverter",
   "value.converter.schemas.enable": "false",

   "transforms":
"extractNewDocumentState,rename,precosAbaixoDe2000Reais,origemSomenteDeSP"
,


"transforms.extractNewDocumentState.type":"io.debezium.connector.mongodb.t
ransforms.ExtractNewDocumentState",
   "transforms.extractNewDocumentState.drop.tombstones": "false",

   "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
   "transforms.rename.regex": ".*",
   "transforms.rename.replacement": "voos.novos",

   "transforms.precosAbaixoDe2000Reais.type":
"io.debezium.transforms.Filter",
   "transforms.precosAbaixoDe2000Reais.language": "jsr223.groovy",
   "transforms.precosAbaixoDe2000Reais.condition": "new
BigDecimal(value.preco.adulto) < 2000 && new
BigDecimal(value.preco.crianca) < 2000",

   "transforms.origemSomenteDeSP.type":
"io.confluent.connect.transforms.Filter$Value",
   "transforms.origemSomenteDeSP.filter.condition": "$.origem[?
(@.aeroporto == 'CGH' || @.aeroporto == 'GRU')]",
```

```
    "transforms.origemSomenteDeSP.filter.type": "include",
    "transforms.origemSomenteDeSP.missing.or.null.behavior": "fail"
}
```

Para instalarmos a atualização, basta rodar o script `./novos-voos-conector-atualizar.sh`. Ao cadastrarmos um novo voo saindo de Congonhas (CGH), uma mensagem é postada no tópico `novos.voos`:

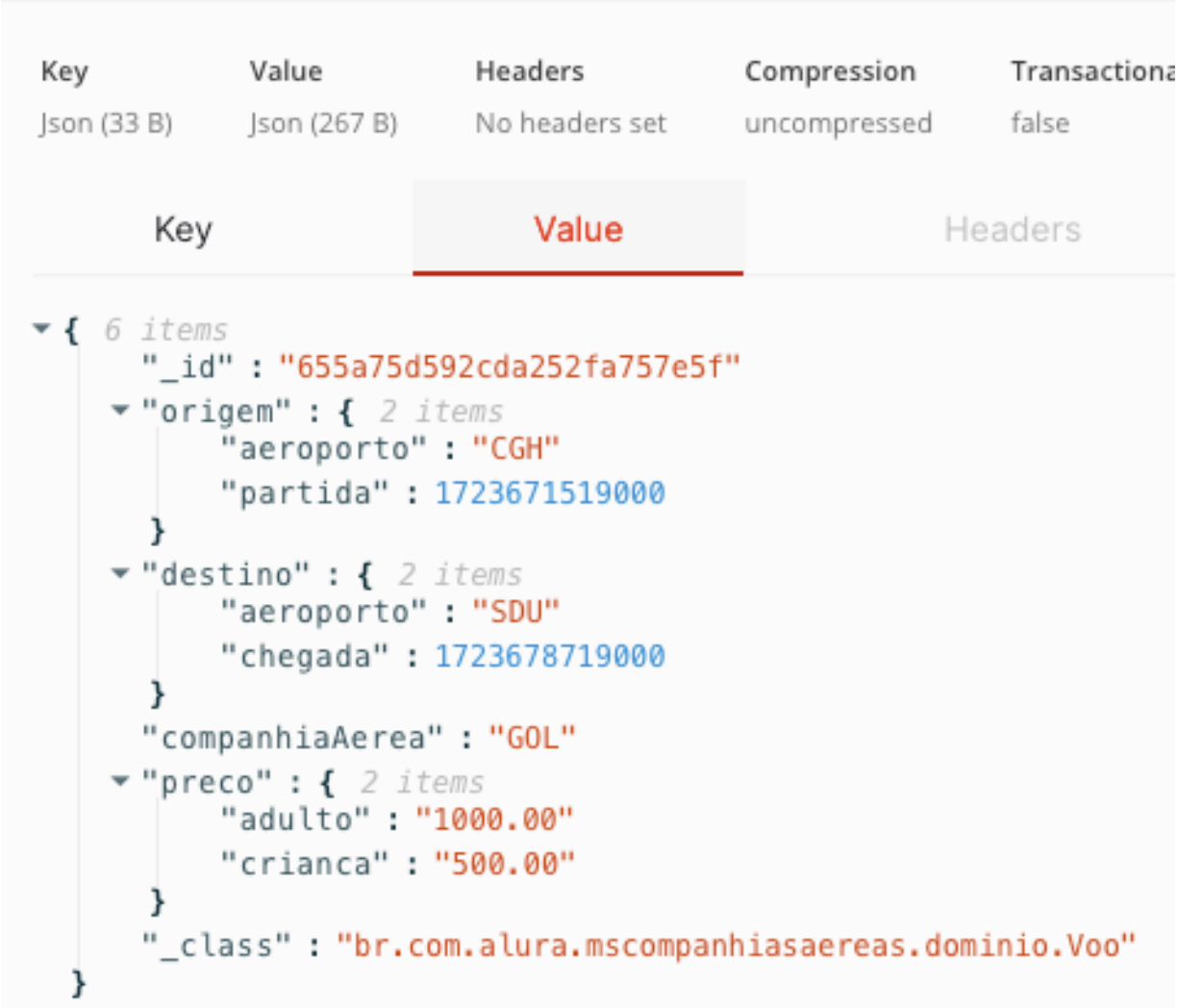


Figura 8.8: Voo saindo de Congonhas postado no tópico sem problemas.

Se cadastrarmos um voo com origem em Guarulhos (GRU), outra mensagem será postada no tópico:

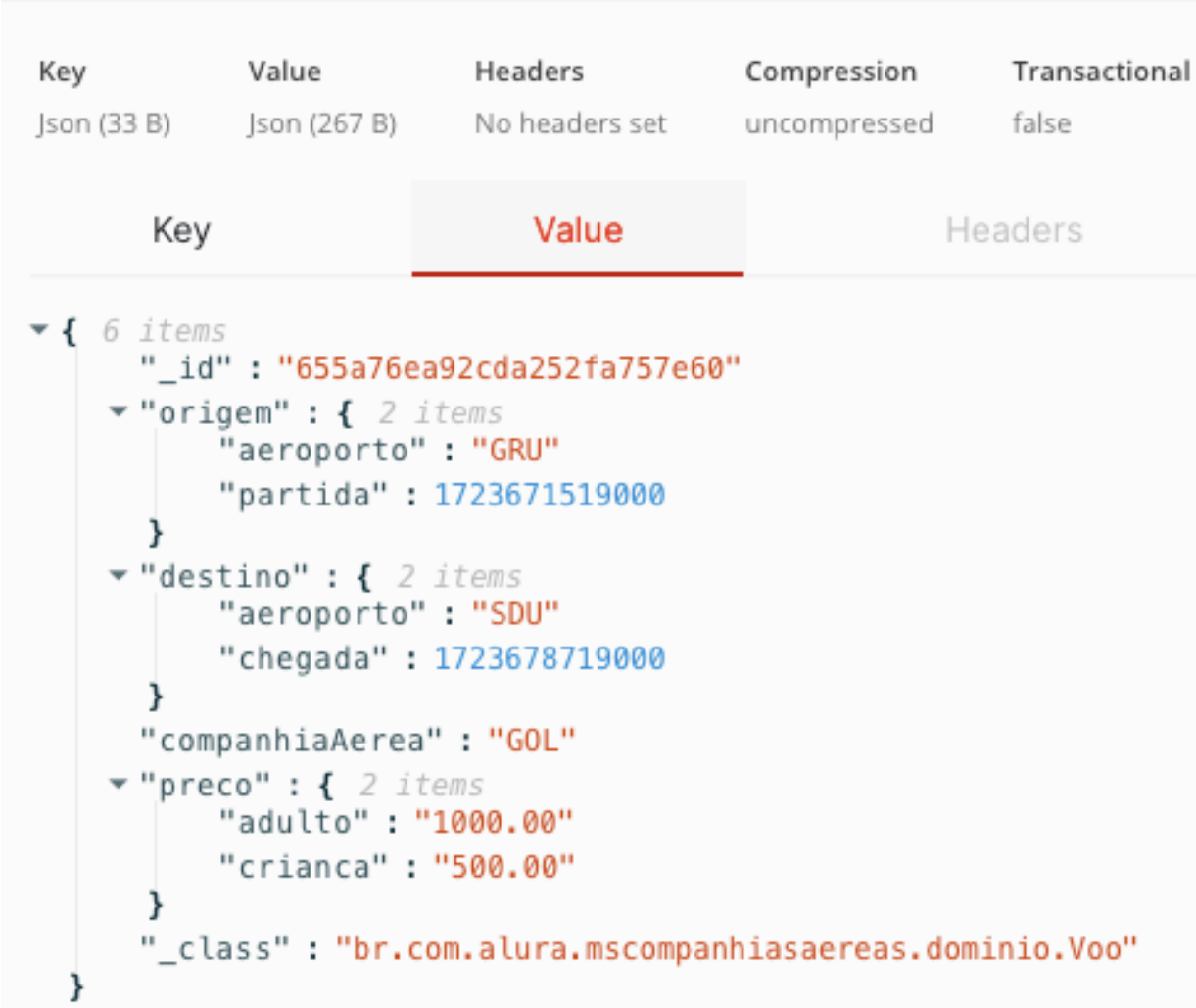


Figura 8.9: Voo proveniente de Guarulhos.

Mas, se tentarmos cadastrar um voo oriundo do aeroporto de Confins (BSB), veremos que nenhuma mensagem é postada no tópico, o que indica que o filtro está funcionando.

> Para mais informações sobre essa SMT, consulte esse link: https://docs.confluent.io/platform/current/connect/transforms/filter-confluent.html

Uma última nota sobre o encadeamento ordenado de SMTs.

8.5 SMTs encadeadas e sua ordenação

Conforme visto neste capítulo e em capítulos anteriores, é possível trabalharmos com mais de uma SMT em um mesmo conector. Basta declararmos, no parâmetro `transforms` todas as SMTs desejadas e em seguida configurá-las, assim como fizemos no arquivo anterior.

Em alguns casos, entretanto, é necessário prestar atenção quanto à ordem de declaração das SMTs no conector. Essa ordem deve estar presente no parâmetro `transforms` e, com isso, as transformações são aplicadas em uma sequência predefinida. Assim, podemos inclusive trabalhar com a ideia de que uma SMT pode depender da transformação realizada por uma SMT anterior.

No nosso caso, poderíamos inserir as transformações `extractNewDocumentState` e `rename` em qualquer ordem, visto que uma não depende da transformação realizada pela outra. Mas, se inserirmos uma das SMTs que inserimos neste capítulo em uma ordem diferente da usada aqui, podemos ter alguns problemas.

Para fins de exemplo, vamos colocar a SMT `precosAbaixoDe2000Reais` como sendo a primeira na cadeia de *transforms*:

```
 "transforms":
"precosAbaixoDe2000Reais,extractNewDocumentState,rename,origemSomenteDeSP"
,
```

Ao atualizarmos o conector, é possível ver que o próximo voo cadastrado lança a seguinte exceção (trechos omitidos por brevidade):

```
[2023-11-20 00:29:20,137] ERROR WorkerSourceTask{id=novos-voos-conector-0}
Task threw an uncaught and unrecoverable exception. Task is being killed
and will not recover until manually restarted
(org.apache.kafka.connect.runtime.WorkerTask)
...
Caused by: javax.script.ScriptException:
```

```
org.apache.kafka.connect.errors.DataException: preco is not a valid field
name
```

Isso ocorre porque a SMT `precosAbaixoDe2000Reais` espera trabalhar com a mensagem já devidamente tratada por SMTs anteriores, removendo *schemas* e outros campos, e mantendo apenas o seu *payload*.

8.6 Resumo

Neste capítulo, fizemos diversas melhorias no projeto, aplicando diversas SMTs, seja do projeto Debezium, seja da própria Confluent. No próximo capítulo, veremos como criar os pacotes de viagem por meio do *ms-pacote* e de ferramentas como o Kafka Streams.

Kafka Streams e algumas topologias avançadas

Vimos diversos conceitos relacionados ao Kafka Connect, como a conexão com diversos bancos de dados e as transformações que podemos realizar nas mensagens antes mesmo de elas chegarem aos tópicos de destino.

Vamos agora realizar voos mais altos, realizando buscas de mensagens em tópicos, filtragens e até mesmo junções de mensagens, por meio do Kafka Streams.

CAPÍTULO 9
Montagem de pacotes de viagem com o ms-pacote e o Kafka Streams

Nos capítulos anteriores, vimos a seleção tanto de voos quanto de hospedagens. Assim, o viajante poderá montar um pacote de viagem composto tanto da passagem de ida quanto do local onde ele poderá se hospedar enquanto ele estiver no local de destino. Chamaremos essa composição de pacote (ou pacote de viagem).

Para que o viajante consiga entender como será montado seu pacote, será criado neste capítulo o *ms-pacote*, que fará a leitura dos tópicos `voos.selecionados` e `hospedagens.selecionadas`, exibindo os valores totais a serem pagos, assim como as informações do voo e da hospedagem selecionadas, para conferência.

> Os códigos aqui apresentados encontram-se na *branch* `cap-9`.

9.1 O ms-pacote

Para que sejam lidas as mensagens dos tópicos, foram montados dois *records* que representam essas mensagens: o `VooSelecionado` e o `HospedagemSelecionada`, ambos presentes no pacote `br.com.alura.mspacote.dominio`:

```
public record VooSelecionado(
    int adultos,
    String cpf,
    int criancas,
    String identificador,
    String destino,
    BigDecimal total,

    @JsonProperty("total_adultos")
    BigDecimal totalAdultos,

    @JsonProperty("total_criancas")
    BigDecimal totalCriancas) {
}

public record HospedagemSelecionada(
    String cnpj,

    @JsonProperty("cpf_comprador")
    String cpfComprador,

    @JsonProperty("valor_adulto")
    BigDecimal valorAdulto,

    @JsonProperty("valor_crianca")
    BigDecimal valorCrianca,
    String identificador,
    String nome
    ) {
}
```

Montamos também o *record* `Pacote`, presente no mesmo pacote, que basicamente recebe instâncias dos *records* acima como parâmetro. Assim que as mensagens de voo selecionado e de

hospedagem selecionada de um mesmo viajante chegarem, o *ms-pacote* consegue montar uma instância do *record* a seguir:

```
public record Pacote(VooSelecionado voo, HospedagemSelecionada hospedagem)
{

    @JsonProperty
    public BigDecimal total() {
        return voo.total().add(hospedagem.total());
    }
}
```

Esse *record* pode ser enviado para um tópico de saída. Assim, outros sistemas podem receber os pacotes de viagem montados e realizar operações, como cobrança ou informar as hospedagens de que um novo hóspede está a caminho. Seguindo as regras descritas no capítulo anterior, vamos chamar esse tópico de `pacotes`.

Com esses *records* prontos, podemos montar dois consumidores simples de mensagens, sendo:

- o *record* `VooSelecionado` para o tópico `voos.selecionados`;
- o *record* `HospedagemSelecionada` para o tópico `hospedagens.selecionadas`.

Para isso, podemos usar o que vimos nos capítulos 5 e 6 e criar classes semelhantes às citadas por lá. Podemos também preparar um produtor de mensagens, que usaria o *record* `Pacote` como representação da mensagem a ser enviada para o novo tópico `pacotes`.

Assim, o fluxo de criação dos pacotes de viagem pode ser representado na figura a seguir:

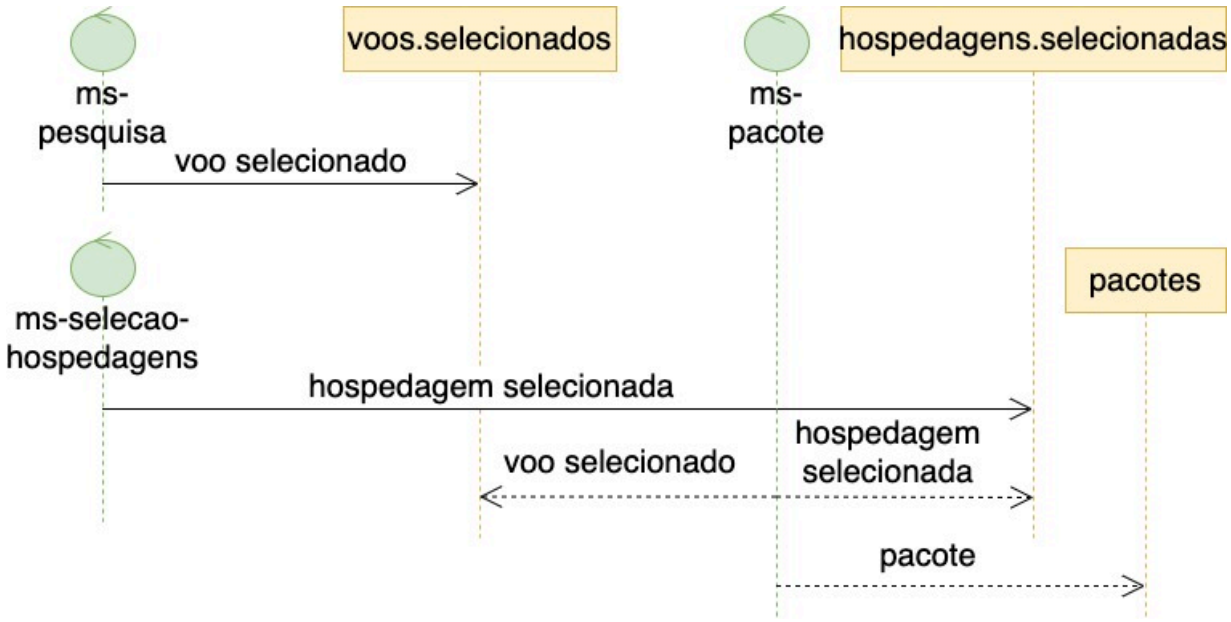


Figura 9.1: Fluxo de criação de pacotes.

Entretanto, teríamos um problema: os microsserviços que enviam voos e hospedagens selecionadas por viajante (respectivamente o *ms-pesquisa* e o *ms-selecao-hospedagens*) podem enviá-los em qualquer sequência. Não é possível garantir que o *ms-pesquisa* sempre envie os voos selecionados antes de o *ms-selecao-hospedagens* enviar as hospedagens selecionadas, e vice-versa.

Felizmente os sistemas que vão usar o *ms-pesquisa* e o *ms-selecao-hospedagens* oferecem garantias de que as mensagens de voos e hospedagens serão enviadas praticamente ao mesmo tempo. Dessa forma, podemos entender que:

- as mensagens não chegarão em uma ordem específica no *ms-pacote*;
- as mensagens chegarão praticamente ao mesmo tempo no *ms-pacote*.

Torna-se necessária alguma solução que garanta o recebimento dessas mensagens e a criação do pacote, independente da ordem que elas chegaram ao *ms-pacote*. Temos, portanto, um cenário onde precisamos compor um objeto (no caso, o `Pacote`) a partir de

mensagens vindas de mais de um tópico. E, assim que todas as mensagens estiverem presentes nos seus tópicos, será possível realizar essa composição.

Problemas como esses podem ser resolvidos usando o Kafka Streams. Vamos ver como ele pode nos ajudar no consumo das mensagens, independente da ordem, ajudando inclusive a gerar uma terceira mensagem para o tópico de saída.

9.2 Kafka Streams

O Kafka Streams é uma biblioteca Java que possui como objetivo tratar as mensagens dos tópicos como fluxos de dados. Com ela, é possível:

- realizar operações semelhantes àquelas encontradas nas *lambda expressions* do Java (*filter*, *map*, *flatMap*, *foreach* e outras) com as mensagens nos tópicos;
- realizar operações semelhantes às presentes em bancos de dados (*groupBy*, *join*, *left join*, *outer join*).

> Este livro procurará destacar as operações semelhantes às presentes em bancos de dados. Mas caso queira conhecer as demais, uma bibliografia recomendada é o livro *Apache Kafka e Spring Boot - Comunicação Assíncrona Entre Microsserviços* de Eduardo Santana (2022).
>
> Mais informações sobre as *lambda expressions* podem ser obtidas no livro *Java 8 Prático - Lambdas, Streams e os Novos Recursos da Linguagem* de Paulo Silveira e Rodrigo Turini (2014).

Suporte ao Kafka Streams no ms-pacote

Para incluir o Kafka Streams no *ms-pacote*, foram incluídas as seguintes bibliotecas no arquivo `pom.xml` :

- `kafka-streams`
- `jackson-databind` (necessária, devido ao fato de que nenhuma das demais bibliotecas utilizadas possuírem um manipulador de JSON)
- `spring-kafka` (presente nos demais microsserviços)

> Normalmente, quando escrevemos aplicações Spring Boot, é natural que utilizemos algum dos *starters* (como o `spring-boot-starter-web`). Alguns desses *starters* trazem consigo como dependências a biblioteca Jackson. Entretanto, como estamos utilizando o `spring-boot-starter` (que não possui o Jackson) como dependência, torna-se necessária a inclusão do Jackson no `pom.xml` .

O `pom.xml` foi escrito da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.4</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>
    <groupId>br.com.alura</groupId>
    <artifactId>ms-pacote</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ms-pacote</name>
    <description>Ferramenta de montagem de pacotes de viagem</description>
    <properties>
```

```
    <java.version>17</java.version>
</properties>
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>

    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

</dependencies>

<build>
```

```
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Já vimos os objetos que representam as mensagens vindas dos tópicos. Idem para o objeto de destino. Vamos então à montagem dos objetos para a criação do fluxo, vindos da biblioteca Kafka Streams.

9.3 Montagem do fluxo de criação de pacotes de viagem com o KStream

Para montar o fluxo, precisaremos entender inicialmente quais classes do Kafka Streams usaremos. As abstrações mais usadas no Kafka Streams são o `Serde` e o `KStream`.

`Serde` (pacote `org.apache.kafka.common.serialization`) é uma abreviação de Serializador/Deserializador, e encapsula ambas as funcionalidades em uma única classe. Podemos entender as instâncias de `Serde` como sendo peças que vão acoplar no início e no fim do fluxo.

> Ao produzirmos ou consumirmos mensagens de tópicos Kafka, obrigatoriamente precisamos pensar na utilização de serializadores (no caso de produção) ou de deserializadores (no caso de consumo). A classe `Serde` é a abstração do Kafka Streams que realiza esse passo.

Já o `KStream` contém todas as operações citadas no item anterior, como *joins* ou *filters*, e transforma os tópicos em streams. Algo semelhante ao que fazemos com *streams* no Java Collections:

```
List<String> nomesDePessoas = new ArrayList<>();
Stream<String> fluxo = nomesDePessoas.stream();
```

Para instanciarmos um `Serde`, precisamos de um serializador e um deserializador. Como as mensagens estão no formato JSON, podemos usar, respectivamente, `JsonSerializer` e `JsonDeserializer` e ter ciência do objeto que será manipulado.

No caso dos voos selecionados (que são instâncias da classe `VooSelecionado`), podemos montar um `Serde` da seguinte forma:

```
Serde<VooSelecionado> serdeVooSelecionadoEvent =
        Serdes.serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(VooSelecionado.class));
```

ou, se utilizarmos o *static import* do Java:

```
Serde<VooSelecionado> serdeVooSelecionadoEvent =
        serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(VooSelecionado.class));
```

Podemos montar `Serdes` para os demais objetos (`HospedagemSelecionada` e `Pacote`):

```
Serde<HospedagemSelecionada> serdeHospedagemSelecionadaEvent =
        serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(HospedagemSelecionada.class));
Serde<Pacote> serdePacoteEvent =
        serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(Pacote.class));
```

Para instanciarmos um `KStream`, é comum precisarmos de uma instância de `Serde` e do tópico a ser lido. Mas, diferente dos *streams* no Java Collections, criamos um `KStream` a partir de um objeto da classe `StreamsBuilder`:

```
StreamsBuilder builder = new StreamsBuilder();
```

> É possível criar uma instância de `StreamsBuilder` usando o mecanismo de Beans do Spring (caso ele esteja sendo usado). Mas, devido à sua simplicidade de criação (via construtor sem argumentos) optamos por instanciá-lo dessa forma.

Dessa forma, o `KStream` dos voos selecionados pode ser montado assim:

```
KStream<String, VooSelecionado> streamVooSelecionadoEvent =
    builder.stream(topicoDeVoosSelecionados, Consumed.with(String(),
serdeVooSelecionadoEvent));
```

ou, se usarmos *static imports*:

```
KStream<String, VooSelecionado> streamVooSelecionadoEvent =
    builder.stream(topicoDeVoosSelecionados, with(String(),
serdeVooSelecionadoEvent));
```

onde `topicoDeVoosSelecionados` é uma `String` com o nome do tópico (no caso, `voos.selecionados`). Por último, a interface `Consumed` determina a maneira com que a saída do *stream será gerada*. No caso:

- a chave será no formato `String`, informada pelo parâmetro `Serdes.String()`;
- o valor será no formato determinado pelo `Serde` `serdeVooSelecionadoEvent`, criado com os demais *serdes* (no caso, JSON).

Seguindo a mesma abordagem, vamos montar também um `KStream` também para as hospedagens selecionadas. O código é bem semelhante:

```
KStream<String, HospedagemSelecionada> streamHospedagemSelecionadaEvent =
    builder.stream(topicoDeHospedagensSelecionadas, with(String(),
serdeHospedagemSelecionadaEvent));
```

É possível montar um `KStream` de formas ligeiramente diferentes dessas, devido à sobrecarga do método `stream()` em `StreamsBuilder`. Acesse o Javadoc da classe no link https://kafka.apache.org/34/javadoc/org/apache/kafka/streams/StreamsBuilder.html para mais informações.

Com os *streams* de voos e hospedagens preparados, vamos preparar a topologia necessária para:

- prepararmos o pacote de viagem;
- enviar o pacote para o tópico `pacotes`.

Ao analisarmos as mensagens no Redpanda, veremos que suas chaves são iguais, caso o viajante selecione tanto voos quanto hospedagens:

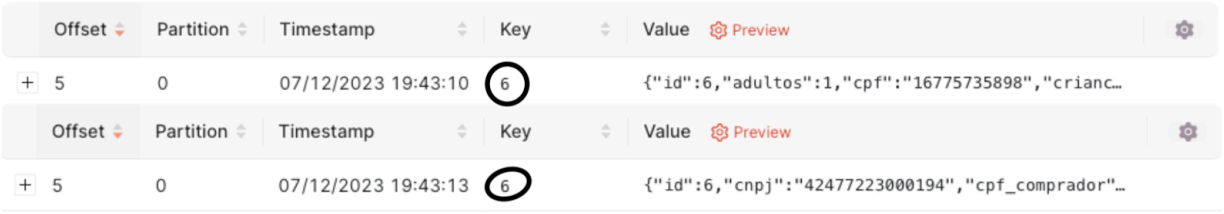


Figura 9.2: Mensagens disparadas praticamente ao mesmo tempo. Notar a igualdade de ids.

Podemos então realizar um *join* das mensagens pelas suas respectivas chaves. O *join* entre dois `KStream` pode ser realizado de diversas maneiras. Aqui, usaremos uma delas que necessita de:

- 2 objetos `KStream`, com um deles realizando o *join*, via método de mesmo nome;
- uma instância da classe `ValueJoiner`. O objeto `ValueJoiner` recebe os objetos que vêm dos streams e realiza a conversão para o objeto desejado;
- uma instância de `JoinWindows`. Esse objeto determina a janela de tempo pelo qual o Kafka Streams vai esperar a chegada da

segunda mensagem. Esse tempo é contado a partir da chegada da primeira mensagem;
- uma instância de `StreamJoined`. Esse objeto determina os *serdes* a serem usados no armazenamento temporário do Kafka Streams. Usado para armazenar o primeiro objeto correspondente à primeira mensagem, enquanto o segundo não chega;
- o tópico de saída, informando quais *serdes* precisarão ser usados para serializar o objeto nesse tópico.

Assim, podemos ilustrar a montagem dos componentes com a figura a seguir:

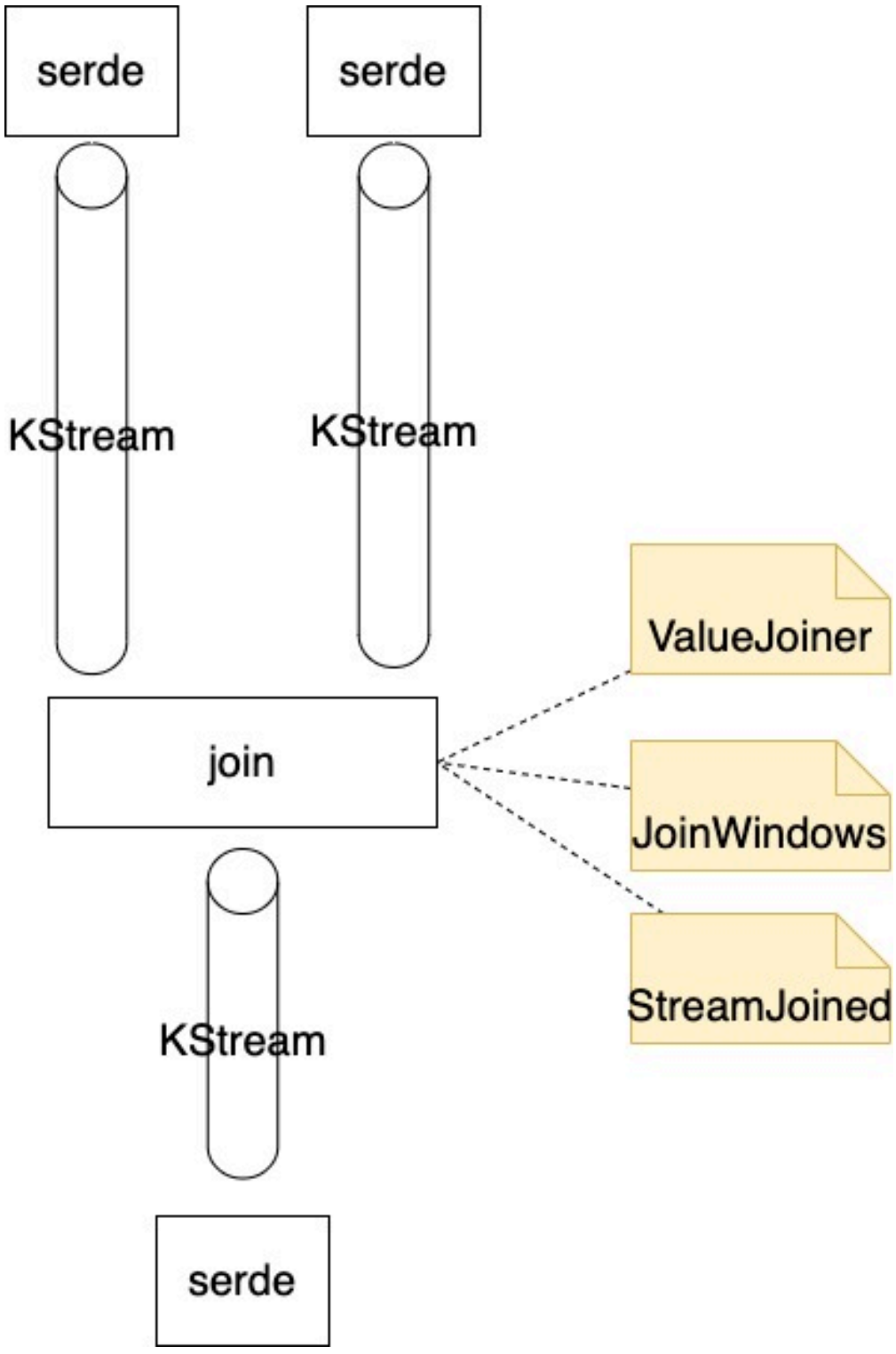


Figura 9.3: Montagem dos objetos necessários para montarmos a topologia para geração de pacotes de viagem.

A figura acima foi inspirada na figura encontrada no final no livro *Enterprise Integration Patterns* (HOHPE, G.; WOOLF, B., 2003). Ela buscou traduzir de maneira didática todos os padrões de integração entre sistemas descritos no livro de forma a simplificar suas localizações em soluções que envolvem troca de mensagens entre aplicações.

O código encontra-se a seguir:

```
streamVooSelecionadoEvent
    .join(streamHospedagemSelecionadaEvent,
        (vooSelecionado, hospedagemSelecionada) -> new
Pacote(vooSelecionado, hospedagemSelecionada),
        JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofSeconds(5)),
        StreamJoined.with(String(), serdeVooSelecionadoEvent,
serdeHospedagemSelecionadaEvent)
    ).to(topicoDePacotesCriados, Produced.with(String(),
serdePacoteEvent));
```

Enquanto temos uma instância de `KStream` sendo usada para realizar o *join*, a outra instância é passada como parâmetro do método `join()`, que recebe outros 3 parâmetros (nesta ordem):

- o `ValueJoiner`, representado por um *lambda expression*;
- a instância de `JoinWindows`;
- a instância de `StreamJoined`.

Logo em seguida, temos o método `to()`, que determina para qual tópico irá a mensagem gerada. E esse método recebe tanto o nome do tópico de saída quanto o `Serde` para serializar a mensagem antes de enviar para esse tópico.

9.4 Problemas com as chaves usadas

As chaves das mensagens dos tópicos são geradas de acordo com o valor do `id` das respectivas tabelas lidas. Com isso, foi possível montarmos uma topologia para a geração de pacotes de viagem.

Mas imagine que, por uma razão qualquer, um viajante decide não selecionar uma hospedagem, mas apenas viajar para um local (ele pode ter um conhecido nesse local, e se hospedar na residência desse conhecido). Nessa situação, nossos microsserviços poderão criar pacotes de viagem contendo viagens de um viajante e hospedagem de outro viajante! Afinal, os *ids* são sequenciais e sem saltos.

Precisamos então pensar em um outro valor a ser inserido como chave das mensagens. E o CPF pode ser um bom substituto, visto que:

- temos uma janela de tempo em que devem ser enviados tanto o voo quanto a hospedagem. Ao chegar uma das 2 mensagens, o tempo começa a contar. A outra mensagem precisa chegar dentro desse tempo. Caso contrário, o pacote não será montado;
- os desenvolvedores dos sistemas que realizam chamadas ao *ms-pesquisa* e ao *ms-selecao-hospedagens* informaram que é impossível um mesmo viajante tentar criar diversos pacotes em um curto espaço de tempo.

Vamos então rever uma das atualizações que aplicamos nos conectores, o `voo-selecionado-conector-atualizar.json`:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
```

```
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"id",
    "decimal.handling.mode": "string",

    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "voos.selecionados"
}
```

Para usarmos o id da tabela `voos_selecionados` como chave, usamos o SMT `ExtractField$Key`, e informamos que queremos usar o campo `id` da mensagem como chave. Seria lógico pensar que é necessário apenas substituir o campo a ser usado como chave, de `id` para `cpf`:

```
"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
"transforms.extractKey.field":"cpf",
```

Mas, ao fazermos isso, o Kafka Connect lança a seguinte exceção:

```
[2023-12-06 22:30:10,294] ERROR WorkerSourceTask{id=voo-selecionado-
conector-0} Task threw an uncaught and unrecoverable exception. Task is
being killed and will not recover until manually restarted
(org.apache.kafka.connect.runtime.WorkerTask)
org.apache.kafka.connect.errors.ConnectException: Tolerance exceeded in
error handler
    at
```

```
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execAnd
HandleError(RetryWithToleranceOperator.java:230)

    ... (trechos omitidos por brevidade)

Caused by: java.lang.IllegalArgumentException: Unknown field: cpf
    at
org.apache.kafka.connect.transforms.ExtractField.apply(ExtractField.java:6
5)
    at
org.apache.kafka.connect.runtime.TransformationChain.lambda$apply$0(Transf
ormationChain.java:50)
    at
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execAnd
Retry(RetryWithToleranceOperator.java:180)
    at
org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execAnd
HandleError(RetryWithToleranceOperator.java:214)
    ... 13 more
```

Esse erro ocorre pois, segundo a documentação do SMT `ExtractField$Key` (presente em https://docs.confluent.io/platform/current/connect/transforms/extractfield.html):

- `ExtractField` extrai um campo de uma chave ou valor complexo (não primitivo, Map ou Struct) e substitui a chave ou valor inteiro pelo campo extraído. Quaisquer valores nulos são passados sem modificação* (tradução livre).

Para entendermos melhor, vamos analisar a chave de uma mensagem gerada sem a aplicação do SMT:

```
Struct{id=5}
```

Podemos ver que o único campo presente é o `id`. Logo, nossa modificação funcionaria apenas se o campo `cpf` estivesse presente na chave.

E se colocássemos o `cpf` na chave da mensagem? É o que veremos a seguir.

9.5 Encadeamento de SMTs ao resgate e o SMT ValueToKey

Além dos diversos SMTs que já vimos, temos também o SMT `ValueToKey` que, segundo sua documentação (presente em https://docs.confluent.io/platform/current/connect/transforms/valuetokey.html):

- Substitui a chave do registro por uma nova chave formada a partir de um subconjunto de campos do valor do registro* (tradução livre).

Esse SMT é utilizado de forma semelhante ao `ExtractField$Key`. Basta inserirmos o SMT e declarar os campos que precisam estar na chave. No nosso caso, é apenas o campo `cpf`:

```
"transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
"transforms.createKey.fields": "cpf",
```

É possível notar que o nome do *transformer* é `createKey`. Só precisamos declará-lo na lista de `transforms` do arquivo de atualização do conector, antes do *transformer* `extractKey`:

```
"transforms": "extractNewDocumentState,createKey,extractKey,rename",
```

A seguir, o arquivo de atualização do conector após essas alterações:

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
```

```
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.server.id": "1234",
    "database.include.list": "pesquisa",
    "table.include.list": "pesquisa.voos_selecionados",
    "skipped.operations": "u,d,t",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:29092",
    "schema.history.internal.kafka.topic": "ms-pesquisa-schemahistory",
    "topic.prefix": "ms-pesquisa",
    "include.schema.changes": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,createKey,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",

    "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "cpf",


"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"cpf",
    "decimal.handling.mode": "string",

    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "voos.selecionados"
}
```

Ao aplicarmos essa atualização e selecionarmos um novo voo (enviando a mensagem adiante para o *ms-pesquisa*):

```
curl -X POST 'http://localhost:8081/voos-selecionados' \
--header 'Content-Type: application/json' \
--data '{
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
```

```
    "adultos": 1,
    "criancas": 2,
    "cpfComprador": "16775735898"
}'
```

podemos ver a mensagem no tópico `voos.selecionados` com a nova chave:

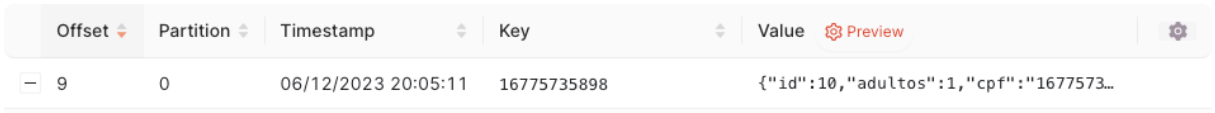


Figura 9.4: Fluxo de criação de pacotes.

> Essa técnica de obtenção de campos específicos da mensagem para usá-los como chaves usando SMTs é também ensinada em https://docs.confluent.io/platform/current/connect/transforms/extractfield.html#chained-transformation

Agora que ajustamos o conector de voos selecionados, precisamos fazer alterações semelhantes no conector de hospedagens selecionadas. Talvez a principal diferença esteja no nome do campo que representa o CPF, que, no caso das hospedagens selecionadas, chama-se `cpf_comprador`.

A seguir, o arquivo de atualização do conector após as alterações:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens_selecionadas",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
```

```
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,createKey,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string",

    "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "cpf_comprador",


"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"cpf_comprador",
    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "hospedagens.selecionadas"
}
```

Ao aplicarmos essa atualização (usando o *script* `hospedagem-selecionada-conector-atualizar.sh`) e selecionarmos uma hospedagem:

```
curl -X POST 'http://localhost:8082/hospedagens-selecionadas' \
--header 'Content-Type: application/json' \
--data '{
    "cnpj": "42477223000194",
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
    "cpf": "16775735898"
}'
```

uma mensagem semelhante à seguinte estará postada no tópico:

Figura 9.5: Fluxo de criação de pacotes.

Ao selecionarmos um voo e uma hospedagem dentro do tempo estipulado na topologia criada no *ms-pacote*, podemos ver o pacote de viagem criado na sequência:

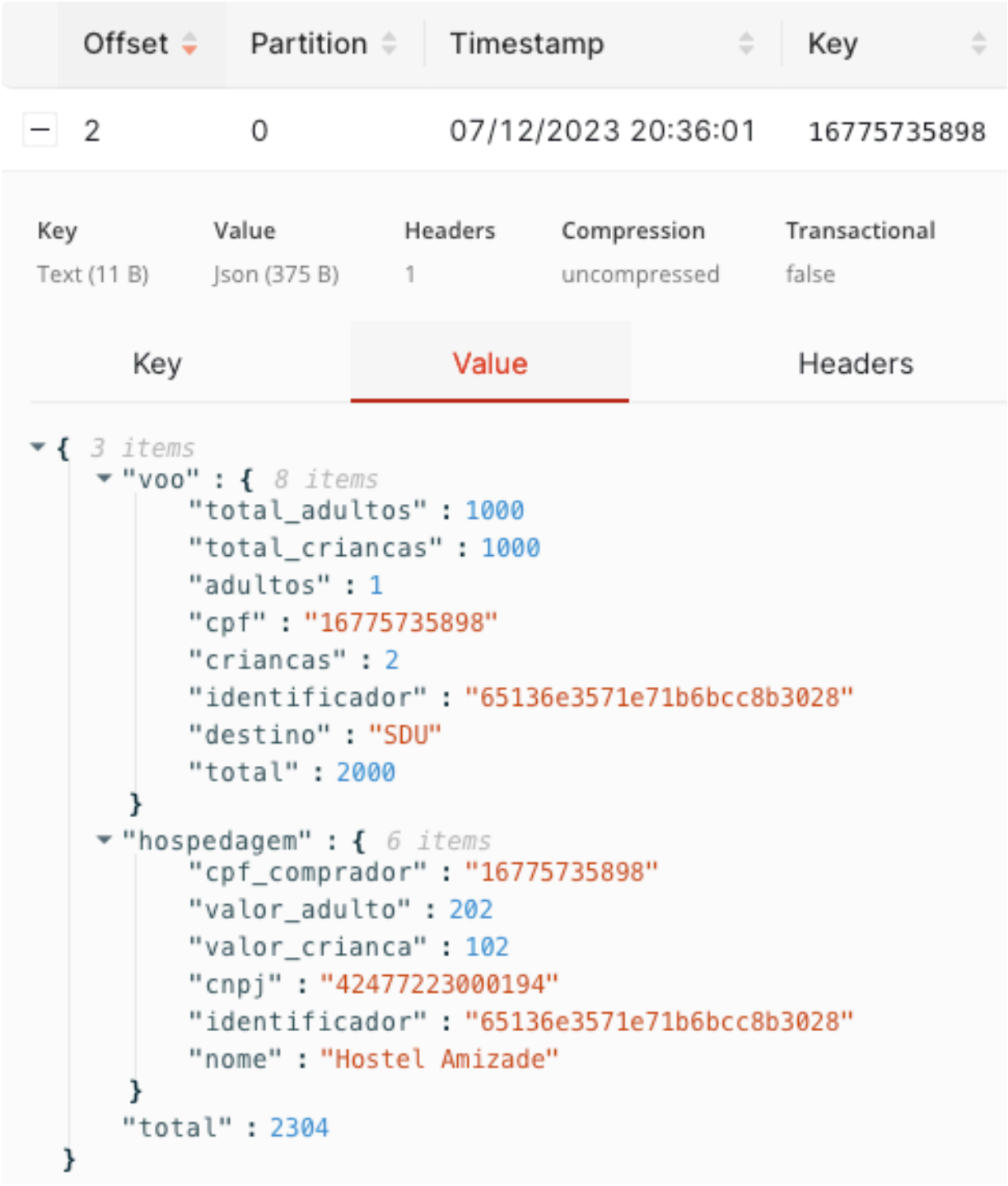


Figura 9.6: Pacote de viagem, com o CPF do viajante como chave.

Repare que não precisamos alterar a topologia que criamos anteriormente, mas apenas os conectores de voos e hospedagens,

visto que tanto o `id` quanto o `cpf` (ou o `cpf_comprador`) são campos de mesmo tipo.

9.6 Resumo

Neste capítulo, começamos a trabalhar com Kafka Streams, realizando o *join* entre mensagens oriundas de dois tópicos diferentes. Também pudemos estudar mais os SMTs, por meio de um exemplo onde encadeamos dois deles para colocarmos um campo da mensagem como sua chave.

No próximo capítulo, veremos como associar os pacotes comprados por um mesmo viajante. E como o Kafka Streams poderá nos ajudar com isso.

CAPÍTULO 10
Informações sobre pacotes vendidos com KTable e o ms-marketing

Como diversas outras empresas, a Alura Viagens e Turismo precisa não somente realizar venda de voos e pacotes aos viajantes, mas precisa saber algumas informações relacionadas às vendas, como o valor total dos pacotes vendidos por hospedagem. Assim, conseguimos entender quais são as hospedagens mais selecionadas, e dar preferência a elas quando o site for exibir as hospedagens disponíveis. É possível obtermos essa informação de algumas maneiras:

- realizando consultas no banco de dados do *ms-selecao-hospedagens*;
- lidando com ciência de dados.

Essa informação precisa estar disponível não somente para nós, mas para empresas de marketing, para que elas possam realizar campanhas com base nessas informações. Afinal, a ideia é destacar as hospedagens que recebem mais viajantes.

Para isso, decidimos utilizar o Kafka e seus tópicos. Mas, para obter essas informações, usaremos mais alguns recursos do Kafka Streams, que veremos neste capítulo.

> Os códigos relacionados ao que veremos aqui estão disponíveis na *branch* `cap-10`.

10.1 Um novo tópico para hospedagens

Como fizemos com todos os tópicos anteriores, precisamos entender como devemos nomear o tópico que armazenará o valor total dos pacotes vendidos por hospedagem.

E para isso, podemos pensar que:

- a principal entidade a ser armazenada é a hospedagem;
- precisamos também armazenar o valor total dos pacotes para a hospedagem.

Podemos entender que:

- a chave desse tópico deve ser o CNPJ da hospedagem;
- o valor deve ser uma mensagem que contenha os dados da hospedagem, acrescida do valor total dos pacotes vendidos.

Há, portanto, uma relação entre `Hospedagem` e `Pacote`, onde uma hospedagem pode conter diversos pacotes. Podemos pensar em um tópico chamado de `hospedagens.pacotes` e também em uma nova entidade, a `PacotesPorHospedagem`, com a seguinte estrutura:

```
public record PacotesPorHospedagem(

    Hospedagem hospedagem,
    BigDecimal valorTotal
    ) {
}
```

Já o *record* `Pacote` pode ser escrito desta maneira:

```
public record Pacote(VooSelecionado voo, HospedagemSelecionada hospedagem)
{

    @JsonProperty
    public BigDecimal total() {
        return voo.total().add(hospedagem.total());
    }
}
```

O método `total()` será muito útil para o cálculo do valor do pacote.

O *record* `Hospedagem` pode ser representado de forma semelhante à sua representação no *ms-pacote*:

```
public record Hospedagem (
        Long id,

        @JsonProperty("aeroporto_mais_proximo")
        String aeroportoMaisProximo,
        String cnpj,

        @JsonProperty("valor_adulto")
        BigDecimal valorAdulto,

        @JsonProperty("valor_crianca")
        BigDecimal valorCrianca,
        String nome,
        String tipo
        ) {
}
```

Com o que definimos, podemos criar uma mensagem por meio do objeto `PacotesPorHospedagem` toda vez que um pacote for criado. O Kafka Streams pode realizar uma associação entre os pacotes criados e as hospedagens existentes no sistema. Mas temos um problema: o Kafka Streams realiza operações com mensagens lidas em tópicos, e não temos um tópico de hospedagens.

Seguindo os conceitos relacionados à nomeação de tópicos, vamos criar um tópico `hospedagens` no nosso `docker-compose.yml` acrescentando uma linha de criação de tópico semelhante às existentes no contêiner `criador-topicos`. Podemos também já aproveitar a ocasião e criar o tópico `hospedagens.pacotes`. As linhas a serem adicionadas serão as seguintes:

```
kafka-topics --create --bootstrap-server kafka:29092 --partitions 1 --
replication-factor 1 --topic hospedagens
kafka-topics --create --bootstrap-server kafka:29092 --partitions 1 --
replication-factor 1 --topic hospedagens.pacotes
```

Precisamos também criar um conector no Kafka Connect para que o tópico `hospedagens` seja populado toda vez que uma nova hospedagem for criada. Ele será semelhante ao conector de `hospedagens.selecionadas`, exceto que:

- a chave do tópico será o `cnpj` da `Hospedagem`;
- o nome do tópico será apenas `hospedagens`, assim como o nome da tabela a que o Kafka Connect vai se conectar.

O conector ficará como adiante:

```
{
    "name": "hospedagem-conector",
    "config": {
        "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
        "database.hostname": "postgres",
        "database.port": "5432",
        "database.user": "postgres",
        "database.password": "abcd1234",
        "database.dbname": "hospedagens",
        "table.include.list": "public.hospedagens",
        "skipped.operations": "u,d,t",
        "topic.prefix": "ms-hospedagens",
        "plugin.name": "pgoutput",
        "value.converter": "org.apache.kafka.connect.json.JsonConverter",
        "value.converter.schemas.enable": "false",
        "transforms":
"extractNewDocumentState,createKey,extractKey,rename",
        "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
        "decimal.handling.mode": "string",

        "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
```

```
        "transforms.createKey.fields": "cnpj",


"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
        "transforms.extractKey.field":"cnpj",
        "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
        "transforms.rename.regex": ".*",
        "transforms.rename.replacement": "hospedagens"
    }
}
```

Com o conector pronto, com todos os microsserviços e a infraestrutura no ar, podemos instalá-lo no Kafka Connect, por meio do *script* `hospedagem-conector.sh`, presente em `docker/cdc-kafka-connect`:

```
./hospedagem-conector.sh
```

Entretanto, ao fazermos isso, o tópico recém-criado `hospedagens` não apresenta nenhuma mensagem.

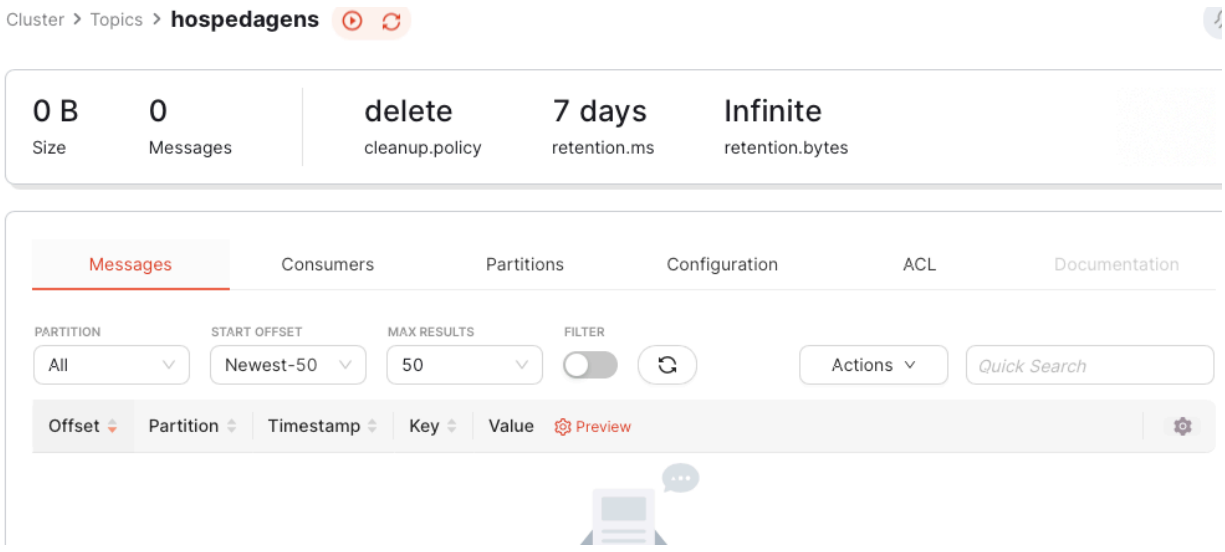
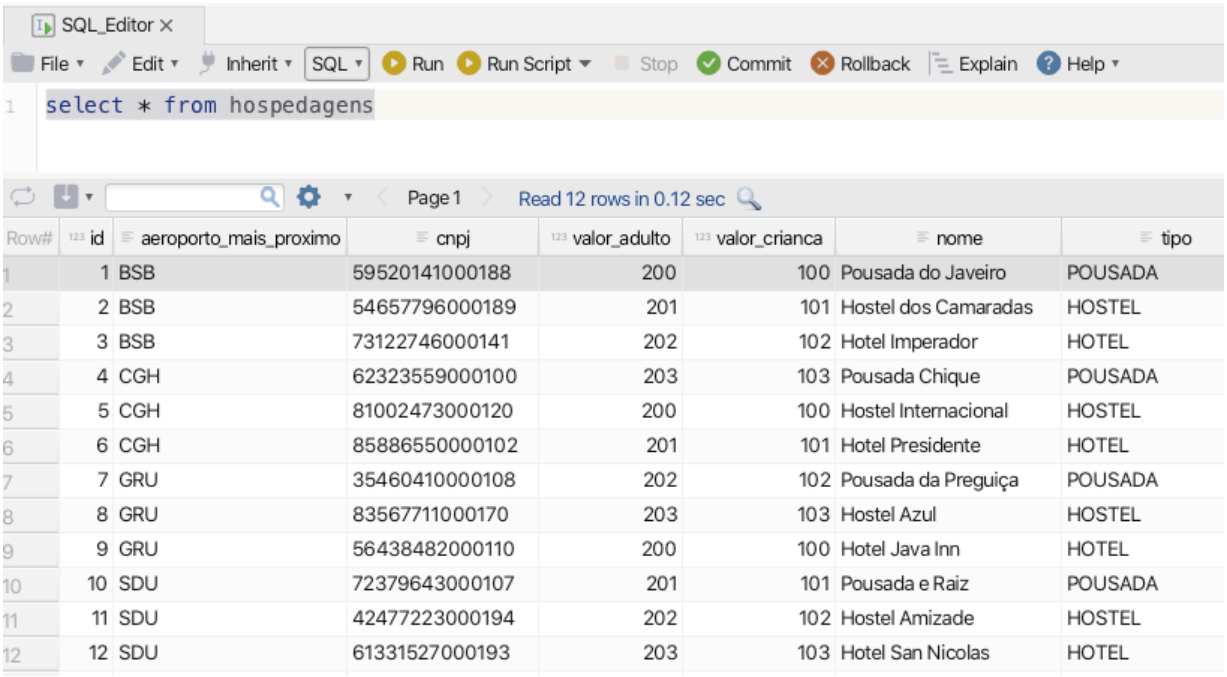


Figura 10.1: Tópico de hospedagens vazio após a instalação do conector.

Mas a tabela de hospedagens contém diversas hospedagens cadastradas:



Row#	id	aeroporto_mais_proximo	cnpj	valor_adulto	valor_crianca	nome	tipo
1	1	BSB	59520141000188	200	100	Pousada do Javeiro	POUSADA
2	2	BSB	54657796000189	201	101	Hostel dos Camaradas	HOSTEL
3	3	BSB	73122746000141	202	102	Hotel Imperador	HOTEL
4	4	CGH	62323559000100	203	103	Pousada Chique	POUSADA
5	5	CGH	81002473000120	200	100	Hostel Internacional	HOSTEL
6	6	CGH	85886550000102	201	101	Hotel Presidente	HOTEL
7	7	GRU	35460410000108	202	102	Pousada da Preguiça	POUSADA
8	8	GRU	83567711000170	203	103	Hostel Azul	HOSTEL
9	9	GRU	56438482000110	200	100	Hotel Java Inn	HOTEL
10	10	SDU	72379643000107	201	101	Pousada e Raiz	POUSADA
11	11	SDU	42477223000194	202	102	Hostel Amizade	HOSTEL
12	12	SDU	61331527000193	203	103	Hotel San Nicolas	HOTEL

Figura 10.2: Tabela de hospedagens.

Esse problema é, na verdade, uma característica não somente do Kafka Connect, mas de todo consumidor Kafka. E será explicada em detalhes a seguir.

O auto-offset-reset e seu papel no Kafka Connect

Quando trabalhamos com consumidores simples Kafka, é comum precisarmos entender como ele deve se comportar em relação a mensagens já existentes no tópico que ele vai consumir. Há as seguintes opções:

- o consumidor poderá ler mensagens no tópico a partir de sua vinculação ao *broker* Kafka, ignorando mensagens anteriores ao seu vínculo;
- o consumidor poderá ler todas as mensagens existentes no tópico, independente do momento de seu vínculo ao *broker* Kafka.

Esse comportamento deve ser configurado no parâmetro `auto-offset-reset` e ele pode possuir os valores:

- `latest`, para a primeira opção citada acima;
- `earliest`, para a segunda opção citada.

Inclusive, por ser um consumidor de tópicos, o *ms-pesquisa* utiliza esse parâmetro no `application.yml`. Como estamos usando Spring Boot, esse parâmetro é chamado de `spring.kafka.auto-offset-reset`. O Kafka Connect também possui esse parâmetro, e ele vem configurado como `latest` por padrão. A diferença é que o Kafka Connect está lendo tabelas e não tópicos.

Para corrigirmos esse comportamento, é necessário inserirmos o parâmetro `consumer.auto.offset.reset` no conector, com o valor `earliest`. E, assim como fizemos nos conectores anteriores, vamos preparar uma atualização para o conector `hospedagens-conector`, que deve conter todos os parâmetros já configurados no arquivo `hospedagem-conector.json`, acrescido do `consumer.auto.offset.reset`. A atualização ficará assim:

```
{
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "abcd1234",
    "database.dbname": "hospedagens",
    "table.include.list": "public.hospedagens",
    "skipped.operations": "u,d,t",
    "topic.prefix": "ms-hospedagens",
    "plugin.name": "pgoutput",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "transforms": "extractNewDocumentState,createKey,extractKey,rename",
    "transforms.extractNewDocumentState.type":
"io.debezium.transforms.ExtractNewRecordState",
    "decimal.handling.mode": "string",
    "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "cnpj",

"transforms.extractKey.type":"org.apache.kafka.connect.transforms.ExtractF
ield$Key",
    "transforms.extractKey.field":"cnpj",
    "transforms.rename.type":
"org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.rename.regex": ".*",
    "transforms.rename.replacement": "hospedagens",

    "consumer.auto.offset.reset": "earliest"
}
```

Ao aplicarmos a atualização, o tópico passará a exibir todas as hospedagens presentes na tabela `hospedagens`:

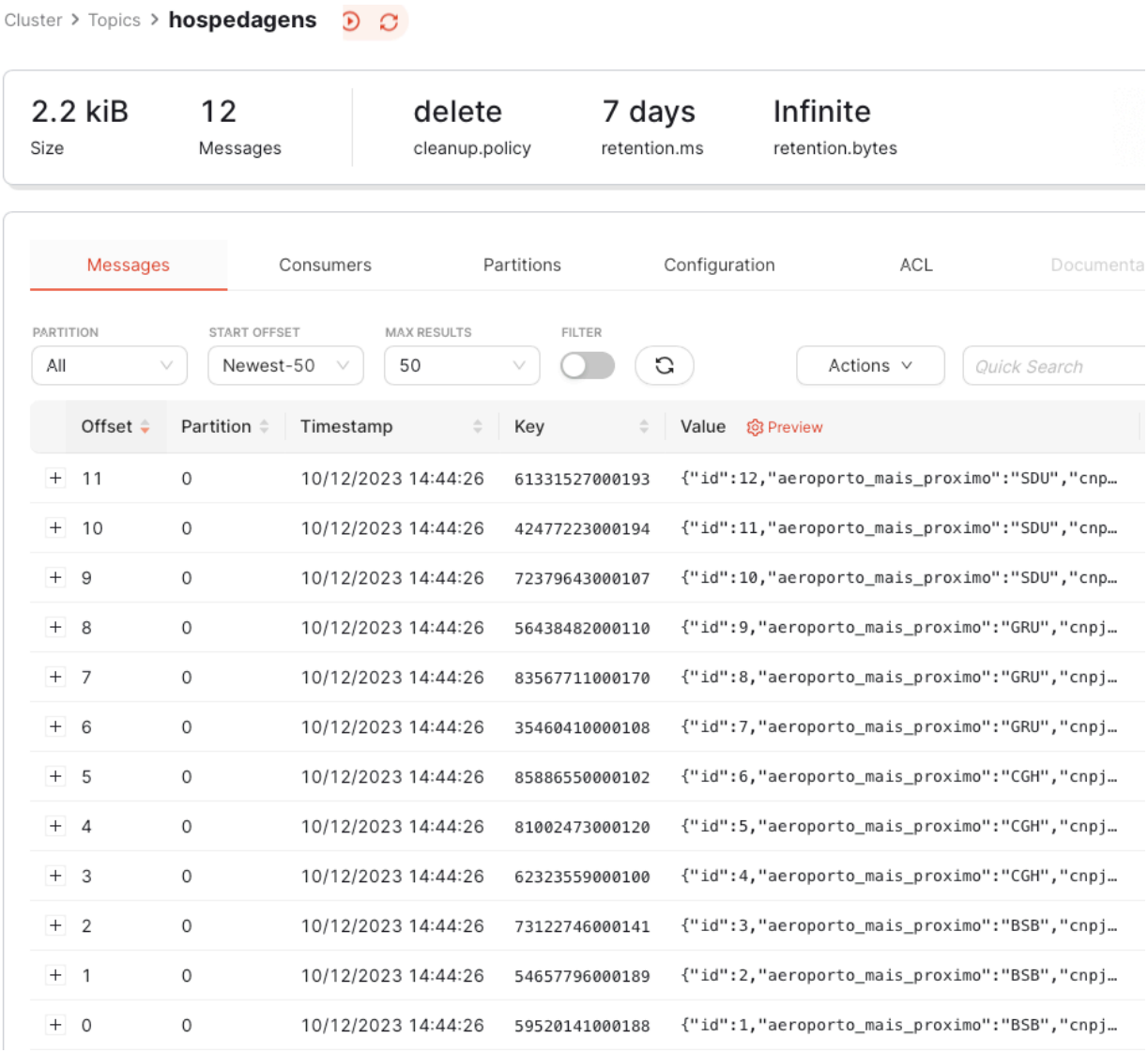


Figura 10.3: Tópico de hospedagens, agora devidamente preenchido com as hospedagens cadastradas anteriormente.

10.2 Criação da topologia e o KTable

Com o tópico `hospedagens` criado e devidamente configurado, podemos começar a preparar a topologia para a criação de mensagens a serem enviadas para o tópico `hospedagens.pacotes`.

Diferente do que fizemos no capítulo anterior, não faremos o *join* entre mensagens pelas chaves dos tópicos, mas entre a chave do tópico `hospedagens` (que armazena o CNPJ das hospedagens) e o campo de mesmo nome presente nas mensagens enviadas ao tópico `pacotes`. Esse *join* está representado na figura a seguir:

Figura 10.4: Relação entre as mensagens dos tópicos de hospedagens e pacotes.

E, assim como fizemos no capítulo anterior, podemos começar pela criação dos *serdes*, sendo um para o tópico de hospedagens, outro para o pacote e o último para os pacotes por hospedagem:

```
Serde<Hospedagem> serdeHospedagemEvent = serdeFrom(new JsonSerializer<>(),
jsonDeserializer(Hospedagem.class));

Serde<Pacote> serdePacoteEvent = serdeFrom(new JsonSerializer<>(),
jsonDeserializer(Pacote.class));

Serde<PacotesPorHospedagem> serdePacotesPorHospedagemEvent = serdeFrom(new
JsonSerializer<>(), jsonDeserializer(PacotesPorHospedagem.class));
```

Para diminuir um pouco a quantidade de código repetido, devemos criar o método `jsonDeserializer()` (usado no código acima):

```
private <T> JsonDeserializer<T> jsonDeserializer(Class<T> clazz){
    return new JsonDeserializer<T>(clazz, false);
}
```

Esse método apenas cria uma instância de `JsonDeserializer` que recebe dois parâmetros: o primeiro representa a classe de objetos que se deseja desserializar; o segundo indica que não precisamos usar os *headers* da mensagem presente no tópico para realizar a desserialização.

Os *headers* das mensagens enviadas podem ser úteis na desserialização. Considere o *header* presente em uma mensagem no tópico `pacotes`:

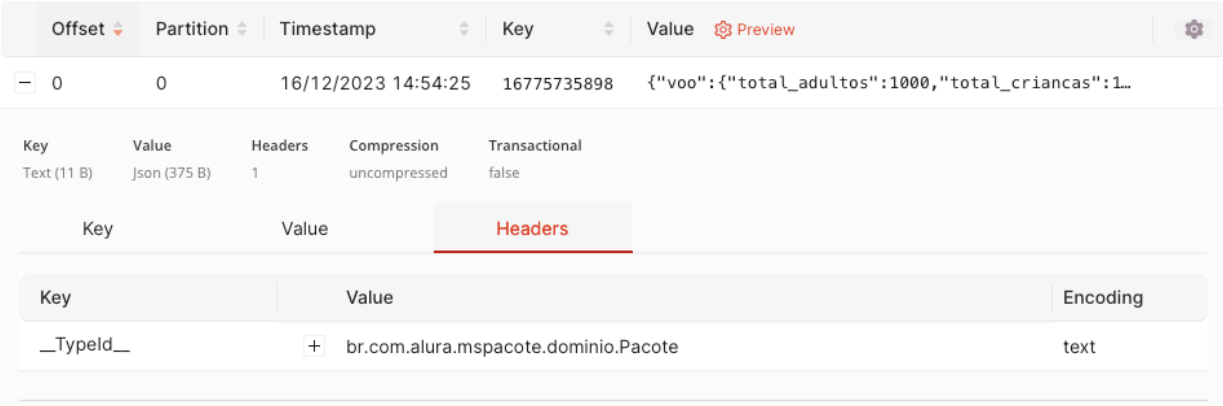


Figura 10.5: Header presente numa mensagem obtida do tópico de pacotes.

Repare que ele contém o nome da canônico da classe em que o produtor se baseou para a criação da mensagem.

Se o segundo parâmetro do construtor de `JsonDeserializer` fosse `true`, essa informação seria usada para informar que o tipo usado pelo produtor da mensagem deve ser levado em consideração. Isso seria ruim no nosso caso, pois os objetos existentes no produtor e no consumidor, embora possuam o mesmo nome e estrutura,

pertencem a pacotes diferentes. Como exemplo, vamos usar a classe `Pacote`:

- ela pertence ao pacote `br.com.alura.mspacote.dominio` no *ms-pacote*;
- e pertence ao pacote `br.com.alura.msmarketing.dominio` no *ms-marketing*.

Logo, para a criação dos objetos `JsonDeserializer` a serem usados pelos nossos *serdes*, usamos o segundo parâmetro do construtor como `false`. Assim, os *headers* das mensagens serão ignorados.

> As mensagens produzidas pelo Kafka Connect não contêm *headers*, pois o Kafka Connect não conhece as classes que originaram as mensagens, mas apenas as fontes de dados.

No mesmo capítulo anterior, utilizamos o `KStream` como objeto para montar a topologia. Objetos `KStream` buscam representar os registros de um tópico como sendo imutáveis e independentes entre si, sem relação entre eles. No caso, precisávamos lidar com voos e hospedagens selecionadas, que não poderiam ser modificadas, mas substituídas. Entretanto, neste capítulo, lidaremos com instâncias do objeto `KTable`.

KTABLE E UMA COMPARAÇÃO COM O KSTREAM

Um `KTable` é uma representação do estado atual de um fluxo de dados, onde as chaves das mensagens representam um papel importante. Para um `KTable`, dadas duas mensagens contendo a mesma chave e num mesmo tópico, será considerada como mensagem a ser utilizada aquela mais recente entre elas.

Exemplo: consideremos o tópico `hospedagens`. Uma hospedagem pode, por exemplo, passar por reajustes de preços nas suas acomodações. Se essa mudança ocorrer, será necessário o envio de uma mensagem ao tópico, com os dados atualizados dessa hospedagem. Mas, para fins de processamento, a mensagem mais recente é a que nos interessa. Logo, se utilizarmos `KTable` nesse tópico para montagem da topologia, o Kafka Streams lerá apenas a mensagem mais recente, ignorando os registros antigos dessa hospedagem. Já se estivéssemos usando `KStream`, todas as mensagens seriam consideradas na topologia, o que pode ser uma boa escolha quando estamos lidando com a seleção de voos e de hospedagens, visto que cada seleção é única.

A montagem de uma instância de `KTable` é bem semelhante à montagem de um `KStream`:

```
KTable<String, Hospedagem> tableHospedagemEvent =
builder.table(topicoDeHospedagens, with(String(), serdeHospedagemEvent));

KTable<String, Pacote> tablePacoteEvent =
builder.table(topicoDePacotesCriados, with(String(), serdePacoteEvent));
```

Para realizarmos o *join* entre os `KTable`, também usamos o método `join()` que, nesse caso, recebe alguns parâmetros diferentes:

- o outro `KTable`;
- uma `Function` que retorne o valor do campo que se pretende fazer *join* com a chave do primeiro `KTable`;

- um `ValueJoiner` que monte o objeto `PacotesPorHospedagem` resultante do *join*.

Um *join* entre dois objetos `KTable` resulta em outro `KTable`. Entretanto, para que possamos enviar o resultado para o tópico de saída, precisamos converter o resultado para `KStream`. Para isso, chamamos o método `toStream()` de `KTable`.

A topologia ficará assim:

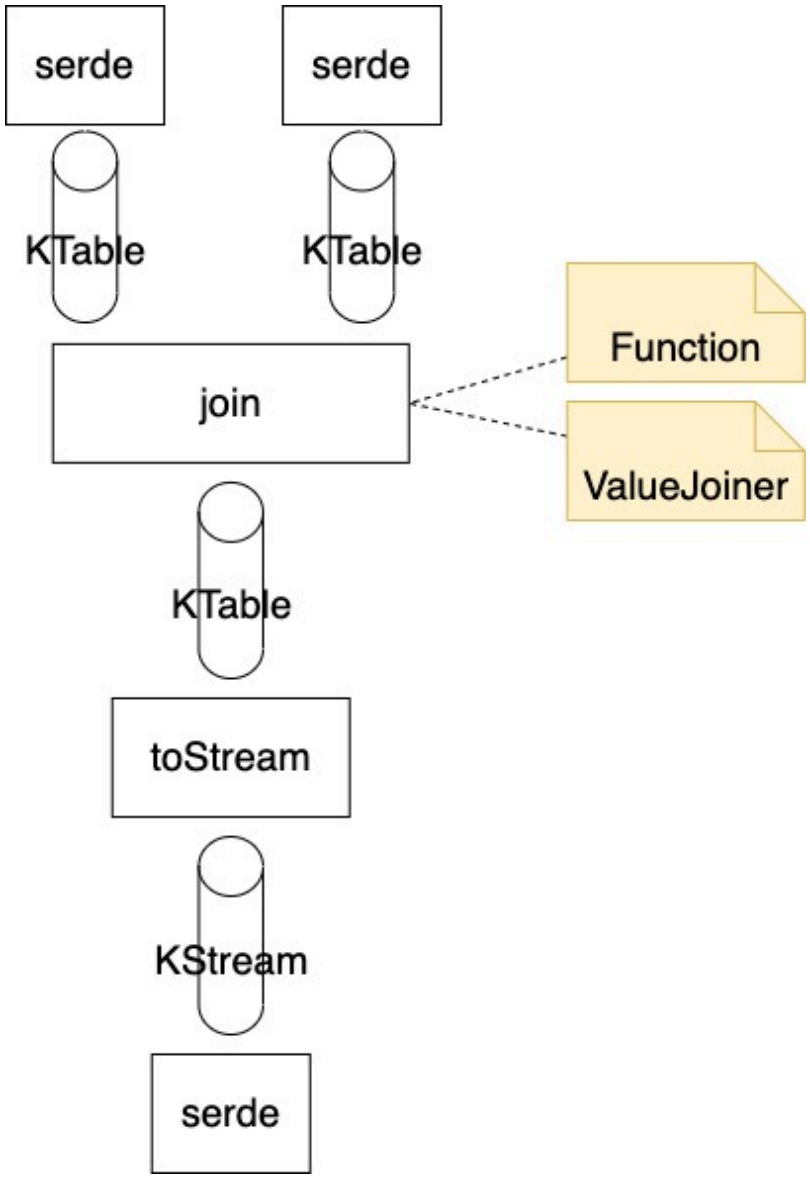


Figura 10.6: Topologia para obtenção de pacotes por hospedagem.

E o código que a representa, da seguinte forma:

```
tablePacoteEvent.join(
        tableHospedagemEvent,
        Pacote::cnpjHospedagem,
        (pacote, hospedagem) -> new PacotesPorHospedagem(hospedagem,
pacote.total()))
    .toStream()
    .to(topicoDePacotesPorHospedagem, Produced.with(String(),
serdePacotesPorHospedagemEvent));
```

Note que é necessária também a criação do método a seguir no *record* `Pacote`, para que a `Function` possa ser chamada na topologia:

```
public String cnpjHospedagem() {
    return hospedagem.cnpj();
}
```

Com essa topologia, ao selecionarmos os voos e uma hospedagem com as seguintes chamadas REST (após subirmos o *ms-pesquisa*, o *ms-pacote*, o *ms-selecao-hospedagens* e o *ms-marketing*):

```
curl -X POST 'http://localhost:8081/voos-selecionados' \
--header 'Content-Type: application/json' \
--data '{
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
    "adultos": 1,
    "criancas": 2,
    "cpfComprador": "16775735898"
}'

curl -X POST 'http://localhost:8082/hospedagens-selecionadas' \
--header 'Content-Type: application/json' \
--data '{
    "cnpj": "42477223000194",
    "idCompanhiaAerea": "65136e3571e71b6bcc8b3028",
```

```
    "cpf": "16775735898"
}'
```

obteremos uma mensagem resultante no tópico `hospedagens.pacotes`, semelhante à seguinte:

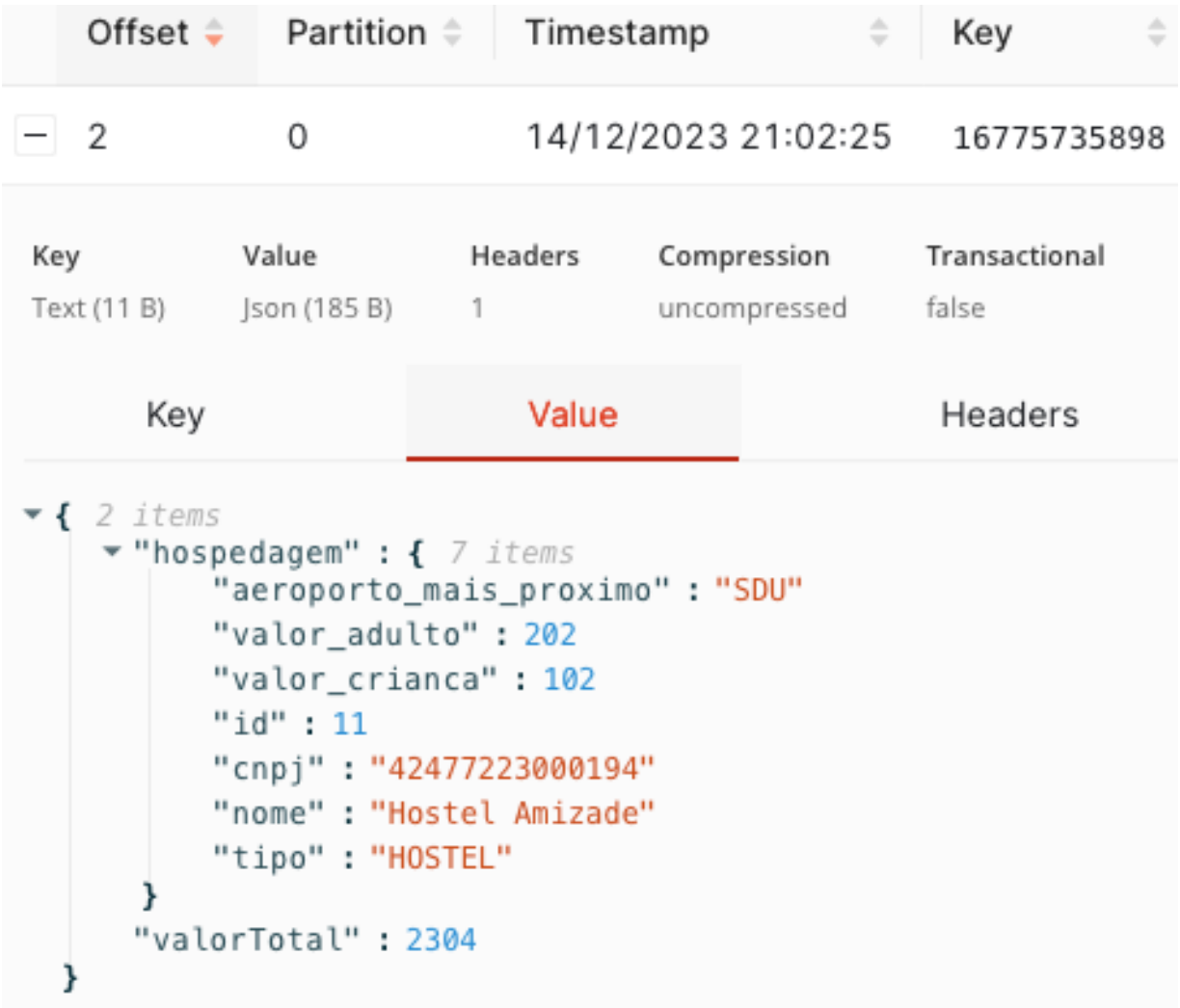


Figura 10.7: Mensagem resultante do join entre hospedagens e pacotes.

É possível notar que a chave da mensagem contém o CPF do viajante. Mas estamos trabalhando com hospedagens e seus pacotes. Logo, faz sentido alterarmos essa chave para uma identificação da hospedagem, no caso, o CNPJ.

Para isso, chamamos o método `selectKey()` do `KTable`, que recebe apenas um objeto do tipo `KeyValueMapper`. Nele, recebemos a chave atual do tópico e uma instância de `Pacote` (visto que o objeto `KTable` manipula pacotes). Assim, nossa topologia passa a ser como a seguinte:

```
tablePacoteEvent.join(
        tableHospedagemEvent,
        Pacote::cnpjHospedagem,
        (pacote, hospedagem) -> new PacotesPorHospedagem(hospedagem,
pacote.total()))
    .toStream()
    .selectKey((chaveAtual, pacote) -> pacote.cnpjHospedagem())
    .to(topicoDePacotesPorHospedagem, Produced.with(String(),
serdePacotesPorHospedagemEvent));
```

E a mensagem resultante após o reinício do *ms-marketing* e as chamadas REST passa a ter o CNPJ da hospedagem como chave da mensagem:

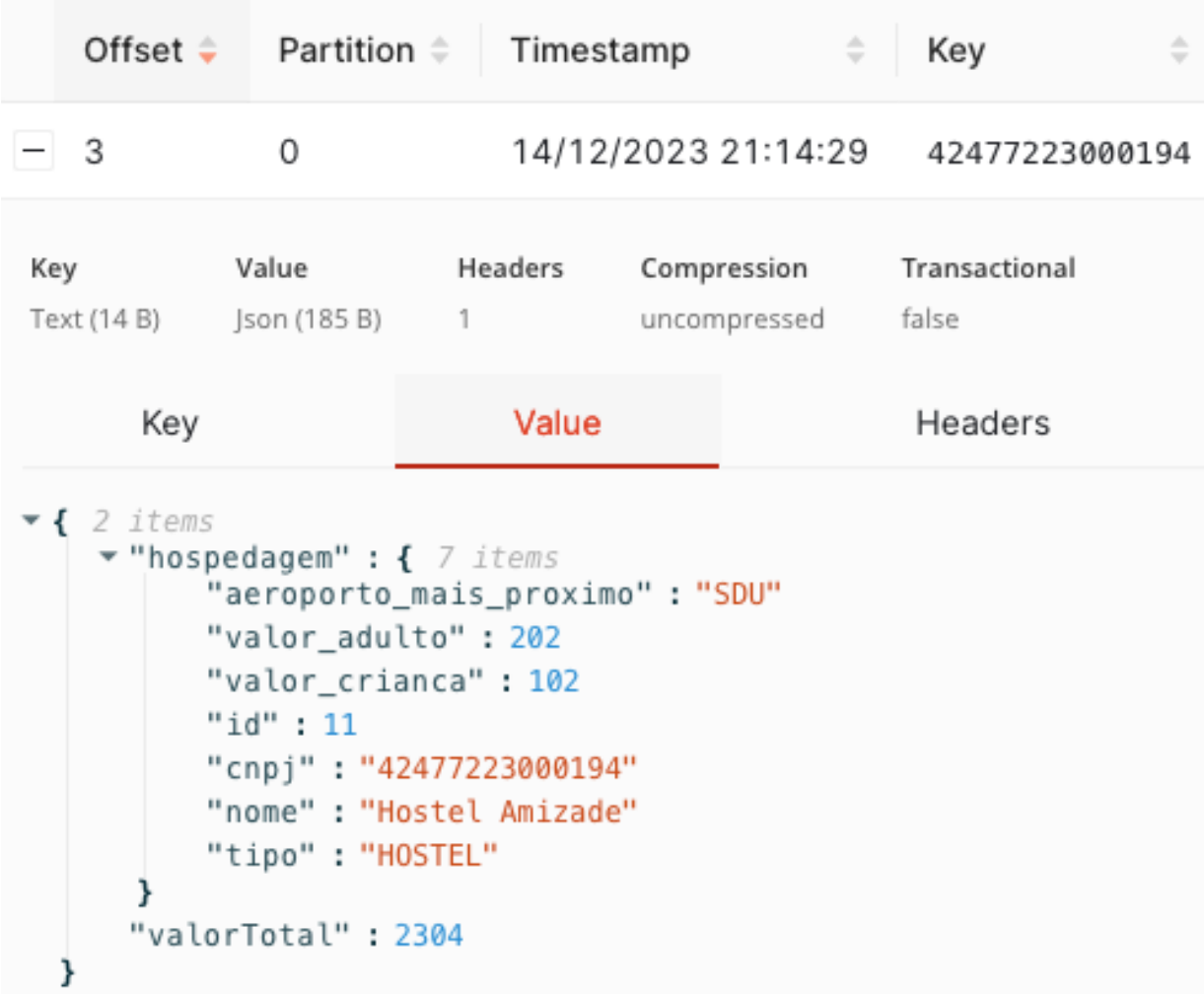


Figura 10.8: Mensagem resultante do join entre hospedagens e pacotes.

Se realizarmos as chamadas REST novamente, veremos que outra mensagem semelhante à anterior foi enviada ao tópico:

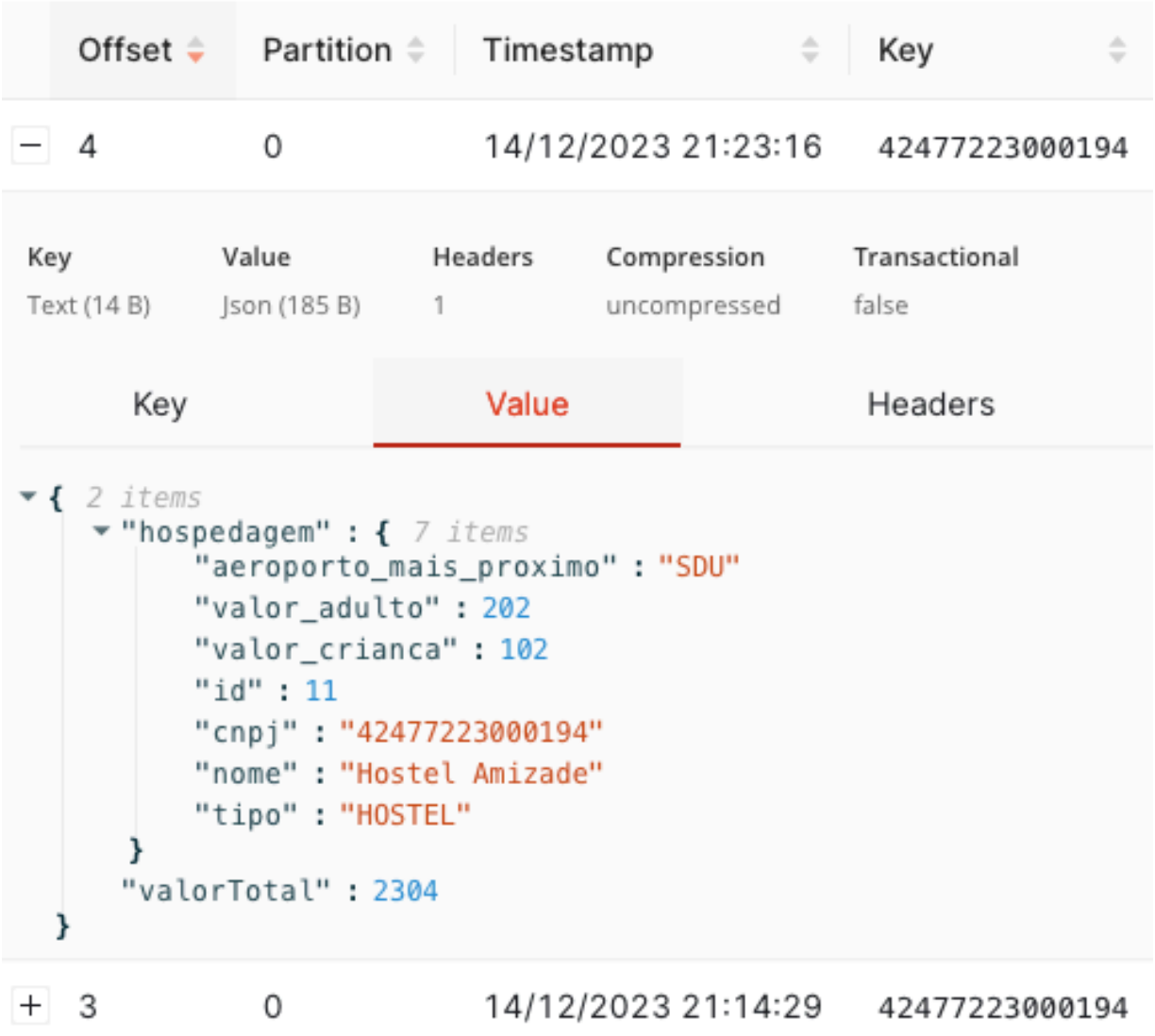


Figura 10.9: Nova mensagem, gerada sem alterações na topologia.

Se quisermos saber o valor total dos pacotes vendidos para uma determinada hospedagem, será necessário obter todas as mensagens relacionadas e somar os seus valores.

Mas podemos inserir essa lógica na nossa topologia, seguindo os passos:

- assim como nos bancos de dados, onde usamos a cláusula SQL `group by`, podemos agrupar as mensagens por CNPJ, usando o método `groupByKey()`, que recebe como argumento um objeto

do tipo `Grouped`. Esse possui características semelhantes aos objetos `Consumed` e `Produced` já vistos anteriormente;
- assim como nas *collections* do Java, podemos realizar operações com as mensagens agrupadas. No caso, podemos somar os valores presentes em cada objeto `PacotesPorHospedagem`.

A topologia final ficará assim com essas alterações:

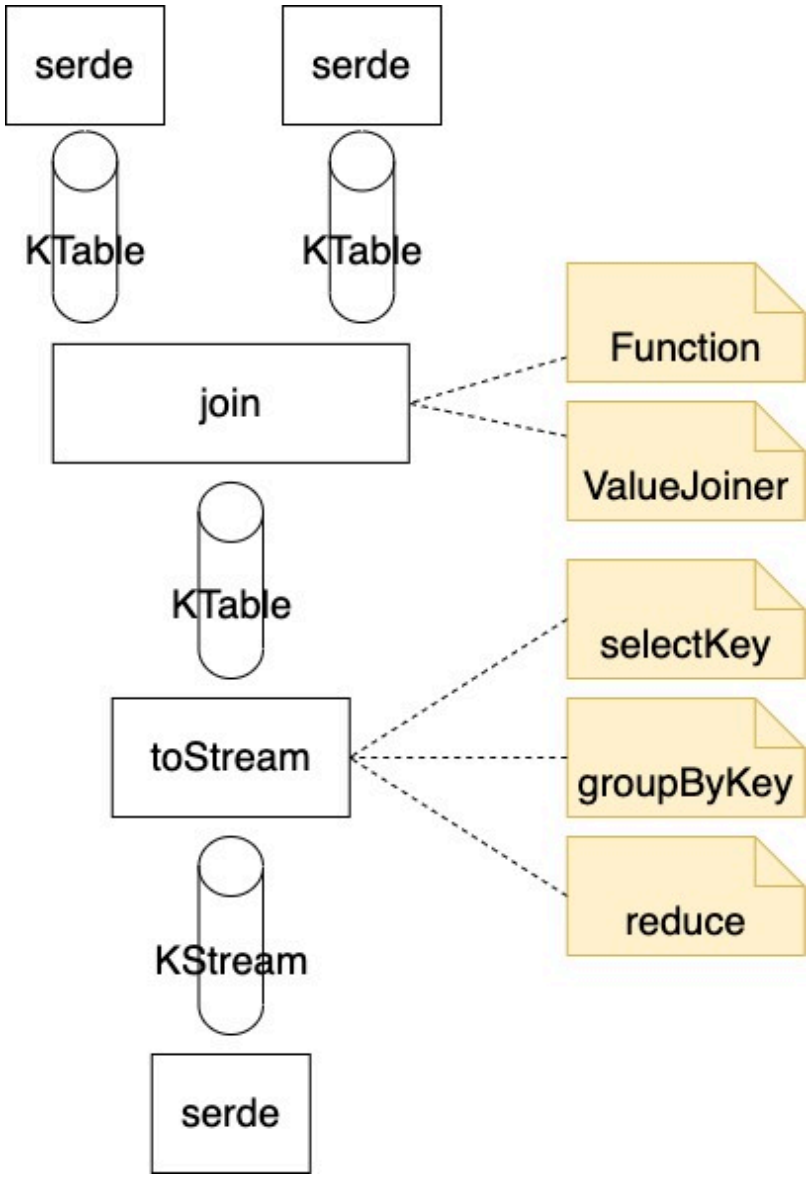


Figura 10.10: Topologia final, após as alterações realizadas.

E o código:

```
tablePacoteEvent.join(
        tableHospedagemEvent,
        Pacote::cnpjHospedagem,
        (pacote, hospedagem) -> new PacotesPorHospedagem(hospedagem,
pacote.total()))
    .toStream()
    .selectKey((chaveAtual, pacote) -> pacote.cnpjHospedagem())
    .groupByKey(Grouped.with(String(), serdePacotesPorHospedagemEvent))
    .reduce((p1, p2) -> new PacotesPorHospedagem(p1.hospedagem(),
p1.valorTotal().add(p2.valorTotal())))
    .toStream()
    .to(topicoDePacotesPorHospedagem, Produced.with(String(),
serdePacotesPorHospedagemEvent));
```

Ao reiniciarmos o *ms-marketing* e realizarmos as chamadas REST novamente, veremos que a mensagem gerada possui um valor total diferente da mensagem anterior. O Kafka Streams passou a somar os valores da penúltima mensagem enviada com os valores da mais recente:

— 5 0 14/12/2023 21:36:

Key	Value	Headers	Compression
Text (14 B)	Json (185 B)	1	uncompressed

Key | Value


```
{ 2 items
  "hospedagem" : { 7 items
      "aeroporto_mais_proximo" : "SDU"
      "valor_adulto" : 202
      "valor_crianca" : 102
      "id" : 11
      "cnpj" : "42477223000194"
      "nome" : "Hostel Amizade"
      "tipo" : "HOSTEL"
    }
    "valorTotal" : 4608
}
```


— 4 0 14/12/2023 21:23:

Key	Value	Headers	Compression
Text (14 B)	Json (185 B)	1	uncompressed

Key | Value


```
{ 2 items
  "hospedagem" : { 7 items
      "aeroporto_mais_proximo" : "SDU"
      "valor_adulto" : 202
      "valor_crianca" : 102
      "id" : 11
      "cnpj" : "42477223000194"
      "nome" : "Hostel Amizade"
      "tipo" : "HOSTEL"
    }
    "valorTotal" : 2304
}
```

Figura 10.11: Mensagens no tópico. Notar a soma dos valores dos pacotes.

10.3 Kafka Streams lento?

É possível que seja notada certa lentidão na execução da topologia acima descrita, dependendo do computador que estiver sendo usado para rodar os exemplos deste livro. Se esse for o caso, pode-se inserir o parâmetro `commit.interval.ms` na lista de parâmetros a serem usados pela topologia. Esse parâmetro é representado pela constante `StreamsConfig.COMMIT_INTERVAL_MS_CONFIG`, e representa o tempo máximo de *commit* das operações efetuadas pela topologia. Seu valor padrão é de 30000 milissegundos, o que faz com que a execução da topologia possa levar um tempo próximo a esse.

Podemos ajustar para um valor muito pequeno, como 1 milissegundo. Assim, os *commits* serão realizados o mais rápido possível. Para isso, podemos atualizar o método `getStreamsConfiguration()` da classe `MontadorDePacotesPorHospedagemKafkaStreams` da seguinte forma:

```
private Properties getStreamsConfiguration() {
    Properties streamsConfiguration = new Properties();
    streamsConfiguration.put(APPLICATION_ID_CONFIG, idDaAplicacao);
    streamsConfiguration.put(BOOTSTRAP_SERVERS_CONFIG, kafkaServerUrl);
    streamsConfiguration.put(DEFAULT_KEY_SERDE_CLASS_CONFIG,
String().getClass().getName());
    streamsConfiguration.put(COMMIT_INTERVAL_MS_CONFIG, "1");

    return streamsConfiguration;
}
```

Para os nossos exemplos, o valor de 1 milissegundo não causará problemas. Entretanto, em aplicações com uma carga muito grande de mensagens sendo processadas pela topologia, valores muito

baixos de tempo de *commit* podem fazer com que o Kafka Streams precise realizar muitos *commits* em um espaço curto de tempo, fazendo com que os ganhos de performance deixem de existir. Logo, é necessário entender qual o melhor valor para esse parâmetro em cada caso.

> Você encontra mais detalhes a respeito desse parâmetro em: https://kafka.apache.org/documentation/#streamsconfigs_commit.interval.ms

10.4 Resumo

Vimos neste capítulo como criar topologias no Kafka Streams usando o `KTable`. Também vimos como realizar melhorias de performance nessa ferramenta.

No próximo capítulo, vamos completar as suítes de testes de nossos microsserviços com a inclusão de testes que lidem com a produção e o consumo de mensagens no Kafka.

Testes automatizados

Vimos diversas maneiras de se trabalhar com Kafka Connect, SMTs e Kafka Streams. Também pudemos observar o funcionamento dessas ferramentas na nossa solução por meio de testes manuais.

Vamos agora fazer um pouso para realizar manutenções preventivas na nossa aeronave. E aproveitar para escrever testes automatizados de produção e consumo de mensagens usando ferramentas suportadas pelo Spring Boot.

Capítulo 11
Testes automatizados de consumo de mensagens

Ao longo deste livro, foram escritos não somente o código dos microsserviços, mas também diversos testes automatizados, que procuraram atestar o funcionamento de todas as classes da aplicação. É possível rodarmos os testes de cada um deles usando um comando semelhante ao seguinte (na pasta `codigo-fonte`):

```
mvn -f <<nome-microsservico>>/pom.xml test
```

onde `<<nome-microsservico>>` é o nome da pasta onde os códigos do microsserviço se encontram.

> Alguns dos testes presentes nos microsserviços são integrados, principalmente com os bancos de dados. Logo, para que eles rodem sem problemas, é importante que a infraestrutura da aplicação esteja no ar. Isso pode ser feito via comando `./infra-up.sh`, localizado na pasta `codigo-fonte/docker`.

A seguir, temos os resultados obtidos rodando o comando acima em cada um dos microsserviços (trechos omitidos por brevidade):

```
mvn -f ms-companhias-aereas/pom.xml test
...
[INFO] Results:
[INFO]
[INFO] Tests run: 17, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  20.590 s
[INFO] Finished at: 2023-12-26T19:39:30-03:00
[INFO] ------------------------------------------------------------------

mvn -f ms-pesquisa/pom.xml test
...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 30, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  34.878 s
[INFO] Finished at: 2023-12-26T19:42:00-03:00
[INFO] ------------------------------------------------------------------

mvn -f ms-selecao-hospedagens/pom.xml test
...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 17, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  25.673 s
[INFO] Finished at: 2023-12-26T19:43:54-03:00
[INFO] ------------------------------------------------------------------
```

```
mvn -f ms-pacote/pom.xml test
...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  9.045 s
[INFO] Finished at: 2023-12-26T19:45:54-03:00
[INFO] ------------------------------------------------------------------

mvn -f ms-marketing/pom.xml test
...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  10.767 s
[INFO] Finished at: 2023-12-26T19:47:41-03:00
[INFO] ------------------------------------------------------------------
```

Vamos então escrever testes automatizados para as classes que lidam diretamente com o `broker` Kafka, seja produzindo ou consumindo mensagens.

11.1 Testes unitários do ReceptorDeVoos no ms-pesquisa

Os códigos exibidos neste capítulo encontram-se na branch `cap-11`.

Conforme vimos em capítulos anteriores, o *ms-companhias-aereas* envia novos voos para o *ms-pesquisa*, assim, a solução pode exibir os voos disponíveis para nossos viajantes. Também envia os voos cancelados, para que o *ms-pesquisa* não exiba voos que não são mais comercializados. Para que isso seja possível, o *ms-pesquisa* possui uma classe `ReceptorDeVoos`, que recebe os novos voos e os cancelados pelas companhias. Vamos revê-la:

```
@Component
@KafkaListener(topics = "${topico.voos.nome}",
        groupId = "${spring.kafka.group_id}",
        containerFactory = "voosKafkaListener")
class ReceptorDeVoos {

    private RepositorioDeNovosVoos repositorioDeNovosVoos;

    @Autowired
    ReceptorDeVoos(RepositorioDeNovosVoos repositorioDeNovosVoos) {
        this.repositorioDeNovosVoos = repositorioDeNovosVoos;
    }

    @KafkaHandler
    void novoVoo(VooCriadoEvent vooCriadoEvent) {
        System.out.println("Voo recebido: " + vooCriadoEvent);
        repositorioDeNovosVoos.save(vooCriadoEvent.toVoo());
    }

    @KafkaHandler
    void vooCancelado(@Header(KafkaHeaders.RECEIVED_KEY) String idVoo,
                      @Payload(required = false) KafkaNull valorNulo) {
        var identificadorCompanhiaAerea =
IdentificadorCompanhiaAerea.fromJson(idVoo);
        System.out.println("Voo cancelado: " +
identificadorCompanhiaAerea);
```

```
repositorioDeNovosVoos.deleteByIdCompanhiaAerea(identificadorCompanhiaAere
a);
    }
}
```

Precisamos verificar de forma automatizada que essa classe está funcionando corretamente. Como fazer isso? Podemos pensar em uma classe que:

- instancie a classe `ReceptorDeVoos`;
- crie um *mock* de `RepositorioDeNovosVoos`;
- e alguns testes com esses objetos, verificando o uso de `RepositorioDeNovosVoos` por `ReceptorDeNovosVoos`.

Essa classe seria mais ou menos assim:

```
class ReceptorDeVoosUnitTest {

    private ReceptorDeVoos receptorDeVoos;
    private RepositorioDeNovosVoos repositorio;

    @BeforeEach
    void setUp(){
        repositorio = mock(RepositorioDeNovosVoos.class);
        receptorDeVoos = new ReceptorDeVoos(repositorio);
    }

    @Test
    void receberNovoVoo(){
        var vooCriado = ObjetosParaTestes.vooCriadoEvent();
        receptorDeVoos.novoVoo(vooCriado);
        verify(repositorio).save(vooCriado.toVoo());
    }

    @Test
    void cancelarVooCriadoAnteriormente(){
        String idCompanhiaAerea = "{\"id\":\"65136e3571e71b6bcc8b3021\"}";
        receptorDeVoos.vooCancelado(idCompanhiaAerea, KafkaNull.INSTANCE);
```

```
verify(repositorio).deleteByIdCompanhiaAerea(IdentificadorCompanhiaAerea.f
romJson(idCompanhiaAerea));
    }
}
```

Esse teste passa sem problemas. Podemos usar o comando a seguir para verificar isso:

```
mvn -f ms-pesquisa/pom.xml test -Dtest="ReceptorDeVoosUnitTest"
...
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running br.com.alura.mspesquisa.infra.kafka.ReceptorDeVoosUnitTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for
boot loader classes because bootstrap classpath has been appended
Voo recebido:
br.com.alura.mspesquisa.infra.kafka.event.VooCriadoEvent@55736208
Voo cancelado:
IdentificadorCompanhiaAerea[identificador=65136e3571e71b6bcc8b3021]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
3.512 s - in br.com.alura.mspesquisa.infra.kafka.ReceptorDeVoosUnitTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  16.530 s
[INFO] Finished at: 2023-12-26T20:39:16-03:00
[INFO] ------------------------------------------------------------------
```

Esse teste verifica a interação da classe `ReceptorDeVoos` com seus atributos, cuja verificação pode ser necessária e útil. Mas, apenas por ele, não é possível provar que a classe vai receber as mensagens oriundas do tópico `voos.novos` .

Essa afirmação pode ser provada apenas comentando algumas anotações usadas em `ReceptorDeVoos`, como `@KafkaListener` ou `@KafkaHandler`, como feito a seguir:

```
@Component
//@KafkaListener(topics = "${topico.voos.nome}",
//        groupId = "${spring.kafka.group_id}",
//        containerFactory = "voosKafkaListener")
class ReceptorDeVoos {

    private RepositorioDeNovosVoos repositorioDeNovosVoos;

    @Autowired
    ReceptorDeVoos(RepositorioDeNovosVoos repositorioDeNovosVoos) {
        this.repositorioDeNovosVoos = repositorioDeNovosVoos;
    }

//    @KafkaHandler
    void novoVoo(VooCriadoEvent vooCriadoEvent) {
        System.out.println("Voo recebido: " + vooCriadoEvent);
        repositorioDeNovosVoos.save(vooCriadoEvent.toVoo());
    }

//    @KafkaHandler
    void vooCancelado(@Header(KafkaHeaders.RECEIVED_KEY) String idVoo,
                      @Payload(required = false) KafkaNull valorNulo) {
        var identificadorCompanhiaAerea =
IdentificadorCompanhiaAerea.fromJson(idVoo);
        System.out.println("Voo cancelado: " +
identificadorCompanhiaAerea);

repositorioDeNovosVoos.deleteByIdCompanhiaAerea(identificadorCompanhiaAere
a);
    }
}
```

Os testes continuam passando:

```
mvn -f ms-pesquisa/pom.xml test -Dtest="ReceptorDeVoosUnitTest"
...
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running br.com.alura.mspesquisa.infra.kafka.ReceptorDeVoosUnitTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for
boot loader classes because bootstrap classpath has been appended
Voo recebido:
br.com.alura.mspesquisa.infra.kafka.event.VooCriadoEvent@55736208
Voo cancelado:
IdentificadorCompanhiaAerea[identificador=65136e3571e71b6bcc8b3021]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
4.434 s - in br.com.alura.mspesquisa.infra.kafka.ReceptorDeVoosUnitTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  12.281 s
[INFO] Finished at: 2023-12-26T20:41:59-03:00
[INFO] ------------------------------------------------------------------
```

Precisamos então de testes que verifiquem também a integração com o tópico no Kafka. E veremos como escrevê-los a seguir.

11.2 Testes integrados de consumo de mensagens do Kafka no ms-pesquisa - @EmbeddedKafka

Para verificarmos a integração do `ReceptorDeVoos` com o Kafka, precisamos escrever código que realize os seguintes passos:

- uma mensagem é enviada para o tópico `voos.novos` ;
- essa mensagem é lida pelo `ReceptorDeVoos` ;

- o `RepositorioDeNovosVoos` é chamado, com os parâmetros corretos.

É possível notar que esse tipo de teste é semelhante a testes de integração com bancos de dados: se queremos testar uma classe de nossa aplicação lendo uma ou mais tabelas do banco, podemos criar um teste que insira uma informação, e nossa classe deve ler essa informação corretamente.

Mas, além dessa semelhança, existe outra: podemos realizar testes com bancos de dados em memória, em vez de ler e escrever em um banco de dados de verdade. E podemos fazer o mesmo com o Kafka, usando a anotação `@EmbeddedKafka`.

Inicialmente, vamos escrever uma classe de teste que receba essa anotação:

```
@EmbeddedKafka
class ReceptorDeVoosTest {
}
```

Precisamos também informar na anotação qual o endereço do Kafka a ser acessado. Por padrão, ele vai acessar o que estiver configurado no parâmetro `spring.kafka.bootstrap-servers`. Mas, como precisamos usar um servidor em memória, devemos inserir o endereço do servidor no parâmetro `bootstrapServersProperty` da anotação:

```
@EmbeddedKafka(bootstrapServersProperty = "spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {
}
```

> O padrão citado acima passou a ser usado a partir da versão 3.0.10 do Spring Boot. Mais informações em https://docs.spring.io/spring-kafka/reference/testing.html#:~:text=The%20bootstrapServersProperty%20is

Essa anotação está presente na biblioteca `spring-kafka-tests`. Logo, precisamos inseri-la no `pom.xml` do microsserviço:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <scope>test</scope>
</dependency>
```

Vamos então escrever nosso primeiro teste de consumo de mensagens. No caso, o recebimento de novos voos vindos do *ms-companhias-aereas*:

```
@EmbeddedKafka(bootstrapServersProperty = "spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {

    @Test
    void receberNovoVoo() {
    }
}
```

Precisamos inicialmente de um produtor de mensagens, que vai enviá-las ao tópico, para serem lidas pelo nosso receptor. Esse produtor pode ser tanto uma instância de `KafkaProducer` ou de `KafkaTemplate`, do Spring Kafka. Como o Spring Boot possui algumas facilidades para a construção de objetos vindos de suas bibliotecas, vamos usar o `KafkaTemplate` e seu método `send()` para realizar o envio.

```
@EmbeddedKafka(bootstrapServersProperty = "spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {

    @Autowired
    private KafkaTemplate<String, VooCriadoEvent> vooCriadoKafkaTemplate;

    @Test
    void receberNovoVoo() {
        vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());
```

```
    }
}
```

O objeto `vooCriadoKafkaTemplate` é um *bean* do Spring. Logo, precisa ser instanciado como tal. Essa instanciação encontra-se na classe `KafkaObjectsParaTestes`, em uma alusão à classe `KafkaObjects`, presente nos pacotes de produção:

```
@Configuration
class KafkaObjectsParaTestes {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Bean
    KafkaTemplate<String, VooCriadoEvent> vooCriadoKafkaTemplate() {
        return new KafkaTemplate<>(new DefaultKafkaProducerFactory<>
(produtorConfigs()));
    }

    private Map<String, Object> produtorConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        return props;
    }
}
```

Vamos então inserir uma instância de `ReceptorDeVoos` na nossa classe de teste:

```
@EmbeddedKafka(bootstrapServersProperty = "spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {

    @Autowired
    private KafkaTemplate<String, VooCriadoEvent> vooCriadoKafkaTemplate;
```

```
    private ReceptorDeVoos receptor;

    @Test
    void receberNovoVoo() {
        vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());
    }
}
```

Podemos então invocar o método `novoVoo()` de `ReceptorDeVoos` dentro do nosso teste:

```
@EmbeddedKafka(bootstrapServersProperty = "spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {

    @Autowired
    private KafkaTemplate<String, VooCriadoEvent> vooCriadoKafkaTemplate;

    @Autowired
    private ReceptorDeVoos receptor;

    @Test
    void receberNovoVoo() {
        vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());
        receptor.novoVoo(vooCriadoEvent());
    }
}
```

Entretanto temos um problema: não é o nosso código que invoca o método `novoVoo()`, mas o próprio Spring! Logo, não temos controle sobre sua chamada. Sabemos apenas que ele será chamado em algum momento pelo framework.

Esse comportamento também é encontrado quando precisamos realizar testes em classes *controller* do Spring MVC. Nesse caso, o Spring é que usa as classes do nosso projeto e não o contrário. Para isso, são devidamente configuradas com anotações e parâmetros para receber as solicitações. Por isso, usamos ferramentas como MockMVC para realizarmos testes que envolvam a execução da lógica contida no *controller*, de acordo com a solicitação HTTP realizada. Os testes dos *controllers* dos microsserviços da nossa solução utilizam essa abordagem.

Não sabemos como o Spring Kafka vai chamar o método `novoVoo()` de `ReceptorDeVoos`. Mas podemos verificar se ele chamou esse método com os parâmetros corretos usando alguns recursos da biblioteca Mockito que veremos a seguir.

MOCKITO MUITO ALÉM DE MOCKS

É bem comum usarmos o Mockito para a criação de *mocks*, como no exemplo adiante:

```
var meuObjeto = mock(MeuObjeto.class);
when(meuObjeto.digaOi()).thenReturn("Oi!");
```

Também não é muito difícil encontramos o uso de métodos como `verify()`, `timeout()` ou `times()`:

```
int comEsseParametro = 1;

// verifica se o método foi chamado
verify(meuObjeto).facaAlgo(comEsseParametro);

// verifica se o método foi chamado apenas 1 vez
verify(meuObjeto, times(1)).facaAlgo(comEsseParametro);

// verifica se o método foi chamado em 2 segundos apenas 1 vez
verify(meuObjeto, timeout(2000).times(1)).facaAlgo(comEsseParametro);
```

Mas essa ferramenta possui também outros recursos, como o *spy* e o `ArgumentCaptor`, que usaremos na nossa classe de teste. Diferente de um *mock* (que é um objeto simulado, onde podemos determinar seu comportamento sem que seja necessária uma instância real dele), um *spy* é um objeto real, que pode ser monitorado para verificar seu comportamento e interações com outros objetos durante os testes.

Já um `ArgumentCaptor` nos permite verificar se um argumento passado a um método é o que esperamos ou não. É muito útil quando não temos controle sobre a chamada de algum método, que é exatamente o que temos no nosso teste.

Veremos como usar todos esses recursos no nosso teste, mas, caso queira entender melhor cada um deles, vale a pena

consultar a documentação da biblioteca, que se encontra em https://site.mockito.org/

Os métodos de `ReceptorDeVoos` serão chamados pelo Spring Kafka. Logo, podemos monitorá-los, no sentido de entender se eles foram chamados corretamente, usando os recursos do Mockito. Poderíamos instanciar esse objeto da seguinte forma:

```
private ReceptorDeVoos receptor = spy(ReceptorDeVoos.class);
```

Mas essa classe é um *bean* do Spring, e instanciá-la dessa forma faria com que não tivéssemos os recursos do framework à disposição. Felizmente o Spring possui as anotações `@MockBean` e a `@SpyBean`. São anotações que se integram com o Mockito, e permitem criar *beans* do tipo *mock* ou do tipo *spy*. Assim, vamos substituir a anotação usada em `ReceptorDeVoos`: de `@Autowired` para `@SpyBean`:

```
@SpyBean
private ReceptorDeVoos receptor;
```

Agora precisamos inserir um `ArgumentCaptor` para capturar o que for passado como parâmetro no método `novoVoo()`. O `ArgumentCaptor` é genérico. Logo, precisa ser tipado:

```
@Captor
private ArgumentCaptor<VooCriadoEvent> vooCriadoEventCaptor;
```

O tipo passado é o tipo do valor que se espera receber como parâmetro do método `novoVoo()`.

Agora podemos corrigir o teste para incluir o `vooCriadoEventCaptor` no teste. Repare que usamos o método `capture()` para capturar o objeto passado como parâmetro:

```
@Test
void receberNovoVoo() {
```

```
    vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());
    receptor.novoVoo(vooCriadoEventCaptor.capture());
}
```

Podemos também verificar se o método `novoVoo()` foi chamado, com uma tolerância de tempo, e no máximo uma vez, usando os métodos `timeout()` e `times()`:

```
@Test
void receberNovoVoo() throws ExecutionException, InterruptedException {
    vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());

    verify(receptor,
timeout(5000).times(1)).novoVoo(vooCriadoEventCaptor.capture());
}
```

E verificar em `vooCriadoEventCaptor` se o objeto passado como parâmetro era o correto. Para isso, usamos o método `getValue()`:

```
    @Test
    void receberNovoVoo() {
        vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());

        verify(receptor,
timeout(5000).times(1)).novoVoo(vooCriadoEventCaptor.capture());

assertThat(vooCriadoEventCaptor.getValue()).isEqualTo(vooCriadoEvent());
    }
```

Por último, será que o `repositorioDeNovosVoos`, presente no `ReceptorDeVoos`, está sendo chamado corretamente? Para verificarmos isso, temos duas alternativas:

- injetar um *bean* de `RepositorioDeNovosVoos` na classe de teste e durante a execução do método de teste, verificar se o objeto `Voo` foi persistido corretamente na base;
- criar um *mock* de `RepositorioDeNovosVoos` na classe de teste, e verificar se o método `save()` foi chamado corretamente.

Vamos então incluir o atributo na classe de teste:

```
private RepositorioDeNovosVoos repositorioDeNovosVoos;
```

Por ser de mais fácil e rápida execução, vamos seguir com a segunda alternativa, inserindo a anotação `@MockBean` para injetar o objeto na classe de teste:

```
@MockBean
private RepositorioDeNovosVoos repositorioDeNovosVoos;
```

Então só precisaremos utilizar o Mockito para realizar as devidas verificações na interação do `ReceptorDeVoos` com esse objeto:

```
@Test
void receberNovoVoo() {
    vooCriadoKafkaTemplate.send(topicoNovosVoos, vooCriadoEvent());

    verify(receptor,
timeout(5000).times(1)).novoVoo(vooCriadoEventCaptor.capture());

assertThat(vooCriadoEventCaptor.getValue()).isEqualTo(vooCriadoEvent());
    verify(repositorioDeNovosVoos,
timeout(5000).times(1)).save(vooCriadoEventCaptor.getValue().toVoo());
}
```

Podemos então executar os testes da aplicação. Tudo certo!

```
mvn -f ms-pesquisa/pom.xml clean test
...

[INFO] Results:
[INFO]
[INFO] Tests run: 33, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  51.451 s
```

Precisamos escrever também o teste relacionado ao cancelamento de voos. E podemos usar passos semelhantes e as mesmas técnicas

aplicadas no teste anterior:

- enviar uma mensagem de cancelamento de voo;
- o `ReceptorDeVoos` deve ler essa mensagem;
- o `RepositorioDeNovosVoos` deve ser chamado, com os parâmetros corretos.

O teste de cancelamento de voo pode ser escrito como o que segue, lembrando que:

- o método trabalha apenas com a chave da mensagem, e não com o seu valor;
- a chave da mensagem está no formato JSON.

```
@Autowired
private KafkaTemplate<IdentificadorCompanhiaAerea, KafkaNull>
vooCanceladoKafkaTemplate;

@Test
void cancelarVooCriadoAnteriormente() {
    var identificadorCompanhiaAerea = identificadorCompanhiaAerea();

    vooCanceladoKafkaTemplate.send(topicoNovosVoos,
identificadorCompanhiaAerea, null);

    verify(receptor,
timeout(5000).times(1)).vooCancelado(toJson(identificadorCompanhiaAerea),
null);
    verify(repositorioDeNovosVoos,
timeout(5000).times(1)).deleteByIdCompanhiaAerea(identificadorCompanhiaAer
ea);
}
```

11.3 Testes integrados de consumo de mensagens do Kafka no ms-pesquisa, usando

diretamente o Kafka

A anotação `@EmbeddedKafka` facilita bastante a escrita e a execução de testes, pois não requer uma instância de Kafka no ar para que sejam feitas verificações nas classes que produzirão ou consumirão mensagens. Entretanto, podem ser necessários testes mais realistas, que utilizem um Kafka real:

- quando se necessita verificar a integração com outros sistemas
- quando os testes dependem de configurações específicas do Kafka que não podem ser facilmente replicadas usando essa anotação

Assim, pode ser necessária a escrita de testes que acessem diretamente o Kafka presente em algum ambiente da empresa (desenvolvimento ou produção, por exemplo), ou mesmo instalado localmente, como é o nosso caso. Para isso, podemos aproveitar quase totalmente o teste que criamos anteriormente. Basta que removamos a anotação `@EmbeddedKafka` da classe. Apenas isso é suficiente para fazer com que os testes produzam e consumam mensagens dos tópicos presentes no nosso Kafka local.

Vamos então comentar essa anotação na nossa classe e rodar os seus testes:

```
// ao comentar a linha abaixo, o teste passará a acessar o Kafka real.
// @EmbeddedKafka(bootstrapServersProperty =
"spring.embedded.kafka.brokers")
class ReceptorDeVoosTest {
```

```
mvn -f ms-pesquisa/pom.xml test -Dtest="ReceptorDeVoosTest"
```

```
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  45.442 s
```

Se acessarmos o tópico, veremos que ele possui duas mensagens: uma relacionada ao teste de cadastro de novo voo, e outra sobre a exclusão de um voo:

−	31	0	04/01/2024 18:53:51

Key	Value	Headers	Compression	Transac
Null	Json (208 B)	1	uncompressed	false

Key | Value


```
{ 5 items
  "_id" : "65136e3571e71b6bcc8b3021"
  "origem" : { 2 items
      "aeroporto" : "CGH"
      "partida" : 1723671519000
  }
  "destino" : { 2 items
      "aeroporto" : "SDU"
      "chegada" : 1723678719000
  }
  "companhiaAerea" : "GOL"
  "preco" : { 2 items
      "adulto" : 1000
      "crianca" : 500
  }
}
```

Figura 11.1: Inclusão de novo voo no tópico Kafka real.

Figura 11.2: Exclusão de voo no tópico Kafka.

11.4 Resumo

Neste capítulo, vimos como podemos escrever testes de consumo de mensagens do Kafka, indo desde os unitários simples até os de integração com uma instância de Kafka.

No próximo capítulo, veremos como escrever testes que verificam as topologias do Kafka Streams que escrevemos nos microsserviços *ms-pacote* e *ms-marketing*.

CAPÍTULO 12
Testes de topologias no Kafka Streams e o RocksDB

Nos capítulos anteriores, vimos como utilizar o Kafka Streams para criar novas mensagens em tópicos, usando de conceitos semelhantes aos usados em bancos de dados, como o *join* um para um entre 2 tópicos e o *join* um para muitos. Entretanto, não escrevemos testes para essas topologias.

Então, neste capítulo, veremos possíveis maneiras de escrevê-los, bem como alguns conceitos que facilitam a sua escrita.

> Os códigos presentes neste capítulo encontram-se na branch `cap-12`

12.1 O RocksDB

Como vimos no capítulo 9, o Kafka Streams realiza um *join* entre duas mensagens para formar uma terceira, usando para isso uma janela de tempo. A seguir, parte do trecho principal do código que realiza essa tarefa:

```
@Autowired
void getStream(StreamsBuilder builder) {
    Serde<VooSelecionado> serdeVooSelecionadoEvent =
            serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(VooSelecionado.class));
    Serde<HospedagemSelecionada> serdeHospedagemSelecionadaEvent =
            serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(HospedagemSelecionada.class));
    Serde<Pacote> serdePacoteEvent =
```

```
            serdeFrom(new JsonSerializer<>(), new JsonDeserializer<>
(Pacote.class));

    KStream<String, VooSelecionado> streamVooSelecionadoEvent =
            builder.stream(topicoDeVoosSelecionados, with(String(),
serdeVooSelecionadoEvent));

    KStream<String, HospedagemSelecionada>
streamHospedagemSelecionadaEvent =
            builder.stream(topicoDeHospedagensSelecionadas, with(String(),
serdeHospedagemSelecionadaEvent));

    streamVooSelecionadoEvent
            .join(streamHospedagemSelecionadaEvent,
                    (vooSelecionado, hospedagemSelecionada) -> new
Pacote(vooSelecionado, hospedagemSelecionada),

JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofSeconds(5)),
                    StreamJoined.with(String(), serdeVooSelecionadoEvent,
serdeHospedagemSelecionadaEvent)
            )
            .to(topicoDePacotesCriados, Produced.with(String(),
serdePacoteEvent));
}
```

Isso significa que:

- se uma mensagem chegar primeiro no tópico `voos.selecionados`, o Kafka Streams esperará por 5 segundos a chegada de uma mensagem no tópico `hospedagens.selecionadas` que contenha a mesma chave que a mensagem em `voos.selecionados`, e fará o *join* entre elas;
- se, no entanto, uma mensagem chegar primeiro no tópico `hospedagens.selecionadas`, o Kafka Streams esperará por 5 segundos a chegada de uma mensagem no tópico `voos.selecionados` que contenha a mesma chave que a mensagem em `hospedagens.selecionadas`, e fará o *join* entre elas.

O cenário acima é um tanto simples, apenas para fins de ilustração. Mas imagine que podemos lidar com milhões de mensagens em um curto espaço de tempo. Como o Kafka Streams consegue lidar com um cenário onde ele terá que aguardar um número significativamente grande de mensagens para realizar o *join* entre elas?

Temos duas alternativas para isso:

- armazenamento em memória. Essa pode ser a mais atraente a princípio, visto que a memória é, normalmente, mais rápida que o disco para obtenção de dados. Entretanto, há alguns problemas nessa abordagem:
  - se o microsserviço for reiniciado por algum motivo, os dados serão perdidos;
  - restrições de espaço: dependendo da quantidade e/ou tamanho das mensagens, poderia ser necessário alocar muita memória para a aplicação.
- armazenamento em disco. Nessa abordagem, temos algumas vantagens:
  - durabilidade dos dados. Se o microsserviço falhar por qualquer motivo, os dados estarão disponíveis para uma futura instância dele consumir;
  - manutenção do estado do processamento de *streams*. Como os dados estão em disco, os microsserviços podem retomar o processamento de onde eles pararam.

Parece que o armazenamento em disco é mais vantajoso que o realizado em memória. Ao menos, quando trabalhamos com Kafka Streams. Optou-se por priorizar a resiliência das aplicações em detrimento da rapidez na sua obtenção.

Mas podemos melhorar esse armazenamento, usando ferramentas que já conhecemos, como bancos de dados, por exemplo. Assim, teremos melhoria de performance em operações de leitura e escrita. E os desenvolvedores do Kafka Streams escolheram o RocksDB

(https://rocksdb.org), que é um banco que utiliza persistência chave-valor e escrito em C++.

Assim que a aplicação entra no ar, o Kafka Streams cria um arquivo para o banco em um diretório temporário, e encapsula todas as chamadas a ele. Logo:

- pode ser importante saber em que pasta o banco foi criado, para fins de backup de dados ou outras operações de manutenção do sistema;
- uma vez a aplicação no ar, o Kafka Streams lida com todo o processo de criação e inserção ou leitura de dados do banco, dispensando na maioria das vezes uma intervenção da pessoa programadora.

Como determinar onde o RocksDB será criado?

Se precisarmos determinar onde queremos que o banco seja criado, devemos inserir uma parâmetro adicional como chave no mapa criado no método `getStreamsConfiguration()` da classe `KafkaObjects`: o `StreamsConfig.STATE_DIR_CONFIG`. E o valor deve ser o local desejado.

A seguir, um exemplo da inserção desse parâmetro no *ms-pacote*:

```
@Bean(name =
KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
KafkaStreamsConfiguration getStreamsConfiguration() {
    Map<String, Object> streamsConfiguration = new HashMap<>();
    streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG,
idDaAplicacao);
    streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);
    streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
String().getClass().getName());
    streamsConfiguration.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG,
"1");
    streamsConfiguration.put(StreamsConfig.STATE_DIR_CONFIG, "./kafka-
streams");
```

```
    return new KafkaStreamsConfiguration(streamsConfiguration);
}
```

Se rodarmos o *ms-pacote* com essa configuração, veremos que o microsserviço cria uma pasta `kafka-streams` com uma pasta `ms-pacote` interna, bem como diversos arquivos pertencentes ao RocksDB:

```
rodrigovieirapinto@MBP-de-Rodrigo ms-pacote % ll
total 48
drwxr-xr-x  9 rodrigovieirapinto  staff    288 22 Out 12:25 bin
drwxr-x---  3 rodrigovieirapinto  staff     96  7 Jan 11:00 kafka-streams
-rw-r--r--  1 rodrigovieirapinto  staff  10070 29 Out 12:30 mvnw
-rw-r--r--  1 rodrigovieirapinto  staff   6608 29 Out 12:30 mvnw.cmd
-rw-r--r--  1 rodrigovieirapinto  staff   2028  6 Jan 11:15 pom.xml
drwxr-xr-x  4 rodrigovieirapinto  staff    128 29 Out 12:30 src
drwxr-xr-x  8 rodrigovieirapinto  staff    256  7 Jan 09:15 target
```

Ao entrarmos na pasta `kafka-streams`, podemos ver os diversos arquivos criados:

```
rodrigovieirapinto@MBP-de-Rodrigo ms-pacote % cd kafka-streams
rodrigovieirapinto@MBP-de-Rodrigo kafka-streams % tree
.
└── ms-pacote
    ├── 0_0
    │   ├── KSTREAM-JOINOTHER-0000000005-store
    │   │   ├── KSTREAM-JOINOTHER-0000000005-store.1704630600000
    │   │   │   ├── 000005.log
    │   │   │   ├── CURRENT
    │   │   │   ├── IDENTITY
    │   │   │   ├── LOCK
    │   │   │   ├── LOG
    │   │   │   ├── MANIFEST-000004
    │   │   │   └── OPTIONS-000007
    │   │   └── KSTREAM-JOINOTHER-0000000005-store.1704630660000
    │   │       ├── 000005.log
    │   │       ├── CURRENT
    │   │       ├── IDENTITY
    │   │       ├── LOCK
```

```
│   │       ├── LOG
│   │       ├── MANIFEST-000004
│   │       └── OPTIONS-000007
│   └── KSTREAM-JOINTHIS-0000000004-store
│       ├── KSTREAM-JOINTHIS-0000000004-store.1704630600000
│       │   ├── 000005.log
│       │   ├── CURRENT
│       │   ├── IDENTITY
│       │   ├── LOCK
│       │   ├── LOG
│       │   ├── MANIFEST-000004
│       │   └── OPTIONS-000007
│       └── KSTREAM-JOINTHIS-0000000004-store.1704630660000
│           ├── 000005.log
│           ├── CURRENT
│           ├── IDENTITY
│           ├── LOCK
│           ├── LOG
│           ├── MANIFEST-000004
│           └── OPTIONS-000007
└── kafka-streams-process-metadata
```

Como localizar o RocksDB em qualquer aplicação?

Determinar onde o RocksDB será criado pode ser importante. Entretanto, essa não é uma informação considerada obrigatória para o Kafka Streams. Por padrão, o RocksDB será criado na pasta especificada pela propriedade de sistema `java.io.tmpdir`. Logo, podemos inserir um método na classe `MontadorDePacotesPorHospedagemKafkaStreams` que recupere essa informação:

```
private void exibirLocalizacaoDoRocksDB() {
    System.out.printf("Localização do RocksDB: %skafka-streams\n",
System.getProperty("java.io.tmpdir"));
}
```

Esse método pode ser chamado durante a construção da topologia, por exemplo:

```
@Autowired
void getStream(StreamsBuilder builder){
    exibirLocalizacaoDoRocksDB();

//montagem da topologia
}
```

Ao rodarmos a aplicação, é possível ver que a informação é exibida logo no início do log, da seguinte forma:

```
Localização do RocksDB:
/var/folders/96/pnd8nnc96dsf21ns3cpncy1m0000gn/T/kafka-streams
```

Naturalmente, a pasta será diferente em cada sistema operacional. Logo, se rodarmos a mesma aplicação em diversos sistemas operacionais, cada um poderá mostrar caminhos diferentes.

12.2 Os objetos StreamsBuilderFactoryBean e KafkaStreams

Nos capítulos 9 e 10 vimos que precisamos criar objetos `KafkaStreamsConfiguration` para as topologias das aplicações. Esses e outros objetos presentes no Kafka Streams estão presentes no objeto `StreamsBuilderFactoryBean`. Segundo a documentação, ele é responsável por fabricar o `StreamsBuilder` e controlar o ciclo de vida de seu objeto interno `KafkaStreams`. O `KafkaStreams`, por sua vez, é o objeto essencial do Kafka Streams, pois ele é o responsável por controlar todo o processamento das topologias criadas.

A documentação dos objetos `StreamsBuilderFactoryBean` e `KafkaStreams` encontram-se, respectivamente, em https://docs.spring.io/spring-kafka/api/org/springframework/kafka/config/StreamsBuilderFactoryBean.html e https://kafka.apache.org/34/javadoc/org/apache/kafka/streams/KafkaStreams.html.

Os Javadocs consultados correspondem a versão de Kafka Streams usada nos exemplos do livro.

Para a montagem dos nossos testes, o `StreamsBuilderFactoryBean` é importante, visto que com ele:

- é possível termos o controle de quando o Kafka Streams deve ser iniciado;
- é possível determinarmos quando ele deve ser encerrado;
- podemos também apagar o RocksDB criado nos testes, assim que o teste se encerrar. Dessa forma, testes mais recentes poderão rodar sem que a execução de testes anteriores seja um problema.

Algumas características de um bom teste automatizado são:

- se necessário, preparar o ambiente para a sua execução. Isso pode incluir inserção de dados em banco, em memória ou em arquivos, subida de ambientes com aplicações e outras tarefas;
- executar o teste, invocando o código que precisa ser testado;
- se necessário, realizar uma limpeza no ambiente de execução, de forma que ele fique no estado anterior à sua execução.

Esse é um dos motivos para que bibliotecas como o JUnit possua anotações como `@BeforeEach` e `@AfterEach`:

- métodos com @BeforeEach serão executados antes de cada teste presente em uma classe;
- métodos com @AfterEach rodarão após cada teste presente em uma classe.

Mais informações sobre essas anotações encontram-se em https://junit.org/junit5/docs/current/user-guide/

Os testes que escreveremos neste capítulo usarão essas anotações, bem como a manipulação do contexto do Kafka Streams, usando alguns métodos de `StreamsBuilderFactoryBean`.

12.3 Teste de topologia no ms-pacote

No capítulo 9 fizemos a criação de pacotes de viagem a partir do voo e da hospedagem selecionadas. Vamos então criar testes automatizados para essa topologia.

Seguindo o que vimos no capítulo 11, precisamos determinar uma sequência de ações a serem executadas para realizarmos o teste. E no caso da topologia presente no *ms-pacote*, faz sentido seguirmos esses passos:

- inserir uma mensagem no tópico `voos.selecionados`;
- inserir uma mensagem no tópico `hospedagens.selecionadas`;
- executar a topologia;
- verificar se o tópico `pacotes` recebeu a mensagem esperada.

Podemos também pensar em um outro cenário: e se a hospedagem for enviada antes do voo? Nesse caso, os passos alteram-se um pouco:

- inserir uma mensagem no tópico `hospedagens.selecionadas`;
- inserir uma mensagem no tópico `voos.selecionados`;
- executar a topologia;
- verificar se o tópico `pacotes` recebeu a mensagem esperada.

Vamos então criar a classe de teste, com os métodos ainda que vazios:

```
class MontadorDePacotesKafkaStreamsTest {

    @Test
    void aoEnviarVooEHospedagemDeveCriarPacote() {

    }

    @Test
    void aoEnviarHospedagemEVooDeveCriarPacote() {

    }
}
```

De forma semelhante ao que fizemos no capítulo anterior, podemos criar instâncias de `KafkaTemplate` para criarmos produtores de mensagens de voos e hospedagens. Assim, vamos inserir dois atributos do tipo `KafkaTemplate`, um para cada tipo de mensagem:

```
@Autowired
private KafkaTemplate<String, VooSelecionado> vooKafkaTemplate;

@Autowired
private KafkaTemplate<String, HospedagemSelecionada>
hospedagemKafkaTemplate;
```

Esses objetos podem ser criados numa classe de criação de *beans*, como a `KafkaObjectsParaTestes` :

```
@Configuration
class KafkaObjectsParaTestes {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.group_id}")
    private String groupId;

    @Value("${spring.kafka.auto-offset-reset}")
    private String autoOffsetReset;

    @Value("${topico.pacotes_criados.nome}")
    private String topicoDePacotesCriados;

    @Bean
    KafkaTemplate<String, VooSelecionado> vooKafkaTemplate() {
        return new KafkaTemplate<>(new DefaultKafkaProducerFactory<>
(produtorConfigs()));
    }

    @Bean
    KafkaTemplate<String, HospedagemSelecionada> hospedagemKafkaTemplate()
{
        return new KafkaTemplate<>(new DefaultKafkaProducerFactory<>
(produtorConfigs()));
    }


    private Map<String, Object> produtorConfigs() {
```

```
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        return props;
    }

}
```

Precisamos também iniciar e encerrar o contexto do Kafka Streams para cada um dos testes criados, manipulando uma instância de `StreamsBuilderFactoryBean` que vimos anteriormente. Vamos inseri-la no nosso teste:

```
@Autowired
private StreamsBuilderFactoryBean streamsBuilderFactoryBean;
```

Em seguida, podemos criar métodos com as anotações `@BeforeEach` e `@AfterEach` do JUnit, para manipularmos o contexto do Kafka Streams:

```
@BeforeEach
void setUp() {
    streamsBuilderFactoryBean.start(); // inicia o Kafka Streams
}

@AfterEach
void tearDown() {
    streamsBuilderFactoryBean.getKafkaStreams().close(); // encerra o
Kafka Streams
    streamsBuilderFactoryBean.getKafkaStreams().cleanUp(); // limpa o
contexto
}
```

É necessário também que informemos ao Spring que manipularemos o contexto do Kafka Streams no nosso teste. Para isso, podemos

usar a anotação `@TestPropertySource` e configurar a propriedade `spring.kafka.streams.auto-startup` como `false`:

```
@TestPropertySource(properties = "spring.kafka.streams.auto-
startup=false")
```

Podemos então enviar mensagens para os tópicos, seguindo o cenário proposto para cada um dos testes:

```
@Test
void aoEnviarVooEHospedagemDeveCriarPacote() {
    vooKafkaTemplate.send(topicoDeVoosSelecionados, vooSelecionado.cpf(),
vooSelecionado);
    hospedagemKafkaTemplate.send(topicoDeHospedagensSelecionadas,
hospedagemSelecionada.cpfComprador(), hospedagemSelecionada);
}

@Test
void aoEnviarHospedagemEVooDeveCriarPacote() {
    hospedagemKafkaTemplate.send(topicoDeHospedagensSelecionadas,
hospedagemSelecionada.cpfComprador(), hospedagemSelecionada);
    vooKafkaTemplate.send(topicoDeVoosSelecionados, vooSelecionado.cpf(),
vooSelecionado);
}
```

Com as mensagens enviadas aos tópicos, precisamos verificar se a mensagem de pacote criado foi enviada ao tópico `pacotes`. E, de forma semelhante ao capítulo anterior, será o *framework* (no caso, o Kafka Streams) que enviará a mensagem para o tópico de destino.

Nesse caso, o problema é ainda mais complexo. Diferente do que ocorreu no capítulo anterior, não temos nem uma classe de recepção de mensagens. Uma boa solução seria executar os envios, aguardar um tempo arbitrário para o Kafka Streams executar seu processamento e depois verificar com um consumidor de mensagens se a mensagem esperada está no tópico.

Precisamos, portanto, de um consumidor de mensagens. Podemos montar uma instância de `KafkaConsumer` como um *bean* do Spring, como no código a seguir (presente em `KafkaObjectsParaTestes`):

```
@Bean
KafkaConsumer<String, Pacote> pacoteKafkaConsumer(){
    KafkaConsumer<String, Pacote> consumidor =
            new KafkaConsumer<>(consumerConfigs(Pacote.class));
    consumidor.subscribe(singletonList(topicoDePacotesCriados));
    return consumidor;
}

private Map<String, Object> consumerConfigs(Class<?> ...clazz) {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "grupo-para-consumidor-de-
teste");
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, autoOffsetReset);
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaServerUrl);

    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class.getName());
    props.put(JsonDeserializer.TYPE_MAPPINGS, typeMappings(clazz));

    return props;
}
```

Com o *bean* criado, podemos injetá-lo na nossa classe de teste:

```
@Autowired
private KafkaConsumer<String, Pacote> pacoteKafkaConsumer;
```

Para aguardarmos o tempo arbitrário, podemos usar o método `Thread.sleep()`, que interrompe o programa. Entretanto, essa solução possui alguns problemas. Dependendo da configuração do computador, o tempo de espera pode ser diferente, para mais ou para menos. Consequentemente:

- precisaríamos colocar um tempo suficientemente alto para que o sistema funcionasse em qualquer máquina. E, ainda assim, não haveria garantias quanto a isso;
- em outros casos, o tempo pode ser menor, visto que o computador possui um poder de processamento melhor;
- se usado em testes (como no nosso caso) pode causar lentidão (devido ao tempo suficientemente alto de interrupção) e torná-los frágeis (pois não há garantias de que, dadas as mesmas entradas, teremos sempre as mesmas saídas)

> Ferramentas como Sonar também recomendam não utilizar recursos desse tipo, por motivos semelhantes aos acima. Mais informações a respeito encontram-se em https://rules.sonarsource.com/java/tag/bad-practice/RSPEC-2925/

12.4 Testes assíncronos e a biblioteca Awaitility

Embora a utilização de `Thread.sleep()` seja uma má prática, de qualquer forma precisamos aguardar um tempo para verificar se a mensagem foi gerada no tópico correto. Como fazer isso?

> Essa é uma característica de testes conhecidos como **ASSÍNCRONOS**. São testes onde o resultado não está imediatamente disponível. Exemplos de testes desse tipo são chamadas de API (e a espera por respostas) ou mesmo produção ou consumo de mensagens em filas ou tópicos.

Uma possibilidade seria:

- de tempos em tempos, verificarmos se a mensagem foi enviada ao tópico;
- se a mensagem estiver no tópico, o teste deve passar;
- se, no entanto, dentro de um determinado tempo, a mensagem não for postada no tópico, o teste deve falhar.

Essa é uma abordagem interessante pois, se o código a ser testado estiver correto, o teste passa, sem que seja necessário aguardarmos um tempo fixo para essa verificação. Verifica-se, portanto, de forma proativa, a condição do teste.

É exatamente isso que a biblioteca Awaitility faz. Com ela, é possível montarmos testes que lidam com a incerteza do tempo necessário para que as condições esperadas do teste sejam atendidas de maneira elegante.

Para a utilizarmos, precisamos inicialmente inserir essa biblioteca no `pom.xml`:

```
<dependency>
    <groupId>org.awaitility</groupId>
    <artifactId>awaitility</artifactId>
    <scope>test</scope>
</dependency>
```

O Awaitility trabalha bastante com o conceito de *Fluent Interface*. Logo, suas classes e métodos buscam ser as mais autoexplicativas possível.

Fluent Interface é um estilo de programação que favorece a nomeação de métodos, atributos e classes de forma a tornar a leitura do código o mais fluente possível. Pode-se inclusive utilizar-se de encadeamento de métodos para atingir esse objetivo. Alguns exemplos desse estilo podem ser encontrados quando utilizamos o Mockito, visto que é uma biblioteca que possui classes que favorecem esse estilo de programação.

Utilizando o exemplo do capítulo anterior:

```
var meuObjeto = mock(MeuObjeto.class);
when(meuObjeto.digaOi()).thenReturn("Oi!");
```

podemos entender que a segunda linha de código executa algo como *quando o método do objeto for executado, retorne essa String*.

Outra biblioteca que possui essas características é o AssertJ. A leitura dos *asserts* criados com ela buscam se parecer com uma descrição textual, por exemplo:

```
BigDecimal quinhentosReaisEUmCentavo = new BigDecimal("500.01");
assertThat(hospedagem.valor()).isEqualTo(quinhentosReaisEUmCentavo);
```

que pode ser interpretado como *verificar se o valor da hospedagem é igual a quinhentos reais e um centavo*.

Vale a pena também conferir o artigo de Martin Fowler, disponível em https://martinfowler.com/bliki/FluentInterface.html.

Como exemplo, precisamos verificar se o tópico `pacotes` contém a mensagem que enviamos, dentro de, no máximo, 6 segundos. Para o Awaitility, isso significa escrever código como o seguinte:

```
ThrowingRunnable runnable = //condições esperadas do teste
await().atMost(Duration.ofSeconds(6)).untilAsserted(runnable);
```

Temos apenas uma condição: dadas todas as mensagens presentes no tópico `pacotes`, uma delas deve ser a esperada.

Para listarmos todas as mensagens presentes no tópico, podemos utilizar o objeto `pacoteKafkaConsumer` da classe `KafkaConsumer` que criamos neste capítulo e listar todas as mensagens presentes no tópico:

```
private List<Pacote> mensagensDoTopico(KafkaConsumer<String, Pacote>
consumidor){
    List<Pacote> eventos = new ArrayList<>();
    consumidor.poll(Duration.ofMillis(1000)).forEach(consumerRecord ->
        eventos.add(consumerRecord.value())
    );

    return eventos;
}
```

Com a lista gerada, precisamos apenas verificar se o pacote esperado está presente na lista:

```
assertThat(mensagensDoTopico(pacoteKafkaConsumer)).contains(pacote);
```

A instância de `ThrowingRunnable` deve ser feita da seguinte forma:

```
ThrowingRunnable runnable = () ->
assertThat(mensagensDoTopico(pacoteKafkaConsumer)).contains(pacote);
await().atMost(Duration.ofSeconds(6)).untilAsserted(runnable);
```

Podemos então escrever nosso primeiro teste desta forma:

```
@Test
void aoEnviarVooEHospedagemDeveCriarPacote() {
    vooKafkaTemplate.send(topicoDeVoosSelecionados, vooSelecionado.cpf(),
vooSelecionado);
    hospedagemKafkaTemplate.send(topicoDeHospedagensSelecionadas,
hospedagemSelecionada.cpfComprador(), hospedagemSelecionada);

    ThrowingRunnable runnable = () ->
assertThat(mensagensDoTopico(pacoteKafkaConsumer)).contains(pacote);
```

```
    await().atMost(Duration.ofSeconds(6)).untilAsserted(runnable);
}
```

Mas podemos também seguir as ideias do *Fluent Interface* e escrever um código assim:

```
@Test
void aoEnviarVooEHospedagemDeveCriarPacote() {
    vooKafkaTemplate.send(topicoDeVoosSelecionados, vooSelecionado.cpf(),
vooSelecionado);
    hospedagemKafkaTemplate.send(topicoDeHospedagensSelecionadas,
hospedagemSelecionada.cpfComprador(), hospedagemSelecionada);

    esperarUmTempoAte(() ->
assertThat(mensagensDoTopico(pacoteKafkaConsumer)).contains(pacote));
}
```

E o método `esperarUmTempoAte()` pode encapsular a chamada aos métodos do Awaitility:

```
private void esperarUmTempoAte(ThrowingRunnable runnable){
    await()
        .atMost(Duration.ofSeconds(6))
        .untilAsserted(runnable);
}
```

> Vale a pena consultar o site http://www.awaitility.org para mais exemplos de utilização dessa biblioteca. Ela, assim como o Mockito e o AssertJ, favorece o estilo de programação *fluent interface*.

E o segundo teste deve ficar da seguinte forma:

```
@Test
void aoEnviarHospedagemEVooDeveCriarPacote() {
    hospedagemKafkaTemplate.send(topicoDeHospedagensSelecionadas,
hospedagemSelecionada.cpfComprador(), hospedagemSelecionada);
    vooKafkaTemplate.send(topicoDeVoosSelecionados, vooSelecionado.cpf(),
vooSelecionado);
```

```
    esperarUmTempoAte(() ->
assertThat(mensagensDoTopico(pacoteKafkaConsumer)).contains(pacote));
}
```

A verificação em si é a mesma do teste anterior, mudando apenas a ordem de envio das mensagens aos tópicos de hospedagens e de voos.

12.5 Testes de topologia no ms-marketing

A topologia usada no *ms-marketing* é um pouco diferente da criada para o *ms-pacote*. Em vez de lidarmos com relacionamentos um para um entre mensagens, no *ms-marketing* temos uma hospedagem e muitos pacotes.

A seguir, o trecho principal de código responsável por montar essa topologia:

```
@Autowired
void getStream(StreamsBuilder builder){
    exibirLocalizacaoDoRocksDB();

    Serde<Hospedagem> serdeHospedagemEvent = serdeFrom(new
JsonSerializer<>(), jsonDeserializer(Hospedagem.class));
    Serde<Pacote> serdePacoteEvent = serdeFrom(new JsonSerializer<>(),
jsonDeserializer(Pacote.class));
    Serde<PacotesPorHospedagem> serdePacotesPorHospedagemEvent =
serdeFrom(new JsonSerializer<>(),
jsonDeserializer(PacotesPorHospedagem.class));

    KTable<String, Hospedagem> tableHospedagemEvent =
builder.table(topicoDeHospedagens, with(String(), serdeHospedagemEvent));
    KTable<String, Pacote> tablePacoteEvent =
builder.table(topicoDePacotesCriados, with(String(), serdePacoteEvent));
```

```
    tablePacoteEvent.join(
            tableHospedagemEvent,
            Pacote::cnpjHospedagem,
                (pacote, hospedagem) -> new
PacotesPorHospedagem(hospedagem, pacote.total()))
        .toStream()
        .selectKey((chaveAtual, pacote) -> pacote.cnpjHospedagem())
        .groupByKey(Grouped.with(String(),
serdePacotesPorHospedagemEvent))
        .reduce((p1, p2) -> new PacotesPorHospedagem(p1.hospedagem(),
p1.valorTotal().add(p2.valorTotal())))
        .toStream()
        .to(topicoDePacotesPorHospedagem, Produced.with(String(),
serdePacotesPorHospedagemEvent));
}
```

Para a escrita de testes dessa topologia podemos utilizar das mesmas técnicas e ferramentas que utilizamos nos testes da topologia do *ms-pacote*, como:

- manipulação do contexto do Kafka Streams, usado uma instância de `StreamsBuilderFactoryBean`;
- criação de produtores e consumidores de mensagens, usando respectivamente instâncias de `KafkaTemplate` e `KafkaConsumer`;
- cenários de teste apropriados;
- utilização do Awaitility, devido ao fato de os testes serem assíncronos.

Para essa topologia, foram montados os seguintes cenários de teste:

- envio de um pacote para uma hospedagem;
- envio de dois pacotes para uma mesma hospedagem. Neste caso, espera-se que uma das mensagens contenha a soma dos valores dos dois pacotes.

Podemos então escrevê-los da seguinte forma:

```
@Test
void aoEnviarUmPacoteDeveSerCriadoUmPacotePorHospedagem() {
```

```
    var pacotesPorHospedagem = new PacotesPorHospedagem(hospedagem, new
BigDecimal("12"));

    pacoteKafkaTemplate.send(topicoDePacotesCriados,
pacote.cpfComprador(), pacote);
    hospedagemKafkaTemplate.send(topicoDeHospedagens, hospedagem.cnpj(),
hospedagem);

    esperarUmTempoAte(() ->
assertThat(mensagensDoTopico(pacotesPorHospedagemKafkaConsumer)).contains(
pacotesPorHospedagem));
}

@Test
void aoEnviarDoisPacotesDeUmaMesmaHospedagemOsValoresDevemSerSomados(){
    var pacotesPorHospedagem = new PacotesPorHospedagem(hospedagem, new
BigDecimal("24"));

    pacoteKafkaTemplate.send(topicoDePacotesCriados,
pacote.cpfComprador(), pacote);
    pacoteKafkaTemplate.send(topicoDePacotesCriados,
pacote.cpfComprador(), pacote);
    hospedagemKafkaTemplate.send(topicoDeHospedagens, hospedagem.cnpj(),
hospedagem);

    esperarUmTempoAte(() ->
assertThat(mensagensDoTopico(pacotesPorHospedagemKafkaConsumer)).contains(
pacotesPorHospedagem));
}
```

A criação dos objetos `hospedagem` e `pacote` pode ser realizada antes de cada teste, no método `setUp()`:

```
private Pacote pacote;
private Hospedagem hospedagem;

@BeforeEach
void setUp() {
    var cnpj = "78654566000176";
    pacote = pacoteParaHospedagemComCnpj(cnpj);
```

```
    hospedagem = hospedagemComCnpj(cnpj);

    // demais códigos
}
```

12.6 Resumo

Neste capítulo vimos algumas técnicas e ferramentas para a criação de testes automatizados que verifiquem topologias criadas para o Kafka Streams.

Encerramento

Agora que vimos diversos conceitos relacionados a Kafka Connect (bem como Kafka Streams, diversos SMTs e testes automatizados), vamos entender como os padrões para microsserviços se utilizam dele para serem aplicados.

Estamos chegando ao final de nossa viagem. Esperamos que você tenha tido uma excelente experiência ao nosso lado.

Capítulo 13
Apêndice - Transactional Outbox Pattern (e padrões relacionados)

Os padrões de projeto podem ser descritos como soluções recorrentes a problemas comuns que surgem durante o desenvolvimento de software. Com o tempo e experiência, os desenvolvedores notaram que era não só possível reutilizá-las como também catalogá-las, para que outros profissionais pudessem absorver esse conhecimento.

Talvez o livro mais famoso sobre o assunto seja o *Padrões de Projeto - Soluções reutilizáveis de software orientado a objetos* (1995) de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, mas já vimos neste livro outros livros que tratam de padrões, como o *Microservices Patterns* (2019) do Chris Richardson. Um dos padrões catalogados nesse livro é o *Transactional Outbox*, de que falaremos neste apêndice.

13.1 Outra collection para o ms-companhias-aereas, e transações com MongoDB

No capítulo 8, implementamos uma SMT que permitia que apenas voos com origem de aeroportos de São Paulo deveriam ser comercializados no Alura Viagens e Turismo. Com isso, diversos voos deixaram de ser enviados para o *ms-pesquisa*.

Após reuniões com o time de dados, verificou-se que esse comportamento fez com que alguns relatórios gerados por eles possuíssem informações incorretas. Eles precisam de informações sobre a origem de todos os voos criados no *ms-companhias-aereas*, sem exceção. E estavam coletando essas informações no *ms-pesquisa* (que possui somente voos com origem em São Paulo), o que fez com que diversas origens não aparecessem nos relatórios.

Foi sugerido que o time de dados passasse a coletar informações da *collection* `voos` do *ms-companhias-aereas*. Entretanto, eles informaram que precisavam apenas dos dados de origem dos voos, e não de todos os dados referentes a um voo. Determinou-se então que uma nova *collection* fosse criada, a `origens_relatorio`, com os seguintes campos:

- a sigla do aeroporto
- a data e hora da partida

A classe `OrigemRelatorio` representa um registro dessa `collection`. Seu código está logo a seguir:

```
@Document(collection = "origens_relatorio")
@EqualsAndHashCode(exclude = "id")
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class OrigemParaRelatorio {

    @Id
    private ObjectId id;
    private String siglaAeroporto;
    private LocalDateTime partida;

    private OrigemParaRelatorio(String siglaAeroporto, LocalDateTime
partida){
```

```
        this.siglaAeroporto = siglaAeroporto;
        this.partida = partida;
    }

    OrigemParaRelatorio(Origem origem){
        this(origem.aeroporto().name(), origem.partida());
    }

    public static final OrigemParaRelatorio origemDoVoo(Voo voo){
        return new OrigemParaRelatorio(voo.deOrigem());
    }
}
```

> O segundo construtor será utilizado nos testes automatizados, que veremos mais à frente.

Assim, cada vez que um novo voo for cadastrado, deve-se persistir não somente o voo, mas também sua origem, em uma *collection* separada. Isso pode ser feito no método `cadastrarNovo(Voo voo)` da classe `VooService`:

```
public String cadastrarNovo(Voo voo){
    repositorioDeOrigensParaRelatorio.save(origemDoVoo(voo));
    return repositorioDeVoos.save(voo).getId().toHexString();
}
```

Temos então duas persistências em `collections` diferentes. Logo, é importante que elas estejam em um contexto transacional, pois, caso uma delas falhe, a outra deve falhar também.

Em bancos de dados relacionais, isso pode ser feito adicionando a anotação `@Transactional` do Spring Boot no método acima, fazendo com que ele fique da seguinte forma:

```
@Transactional
public String cadastrarNovo(Voo voo){
    repositorioDeOrigensParaRelatorio.save(origemDoVoo(voo));
```

```
    return repositorioDeVoos.save(voo).getId().toHexString();
}
```

Mas a base de dados do *ms-companhias-aereas* é MongoDB. E por *default*, o Spring Data MongoDB não suporta transações. Logo, precisamos injetar um *bean* de gerenciamento de transação no contexto do Spring. No caso do MongoDB, é o `MongoTransactionManager`, e usaremos a classe a seguir para isso:

```
@Configuration
class MongoConfiguration extends AbstractMongoClientConfiguration {

    @Bean
    public MongoTransactionManager transactionManager(MongoDatabaseFactory
dbFactory) {
        return new MongoTransactionManager(dbFactory);
    }
}
```

> A partir da versão 4 do MongoDB, as transações multidocumentos passaram a ser possíveis. Logo, podemos utilizá-las na nossa aplicação.

O *bean* `MongoDatabaseFactory` (passado como parâmetro no método acima) é montado na própria classe `AbstractMongoClientConfiguration` (pertencente ao *framework* Spring Data MongoDB), e ele recebe 2 parâmetros:

- um objeto `MongoClient` (que contém as credenciais de acesso ao MongoDB).
- uma `String` contendo o nome do banco de dados. Essa informação é obtida via implementação do método `getDatabaseName()`, realizada na nossa classe concreta `MongoConfiguration`.

Adiante, o código que injeta o *MongoDatabaseFactory* no contexto do Spring:

```
@Bean
public MongoDatabaseFactory mongoDbFactory(){
    return new SimpleMongoClientDatabaseFactory(mongoClient(),
getDatabaseName());
}
```

Vamos também sobrescrever o método `mongoClient()` da classe pai, injetando as propriedades de conexão com o banco:

```
@Value("${spring.data.mongodb.uri}")
private String mongodbUri;

@Override
public MongoClient mongoClient() {
    var mongoClientSettings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString(mongodbUri))
            .build();
    return MongoClients.create(mongoClientSettings);
}
```

Para a implementação de `getDatabaseName()` podemos, de forma semelhante ao que fizemos no método anterior, injetar outra propriedade, a `spring.data.mongodb.database` em `MongoConfiguration`, e retornar seu valor nesse método, da seguinte forma:

```
@Value("${spring.data.mongodb.database}")
private String databaseName;

@Override
protected String getDatabaseName() {
    return databaseName;
}
```

A classe `MongoConfiguration` completa encontra-se a seguir:

```
package br.com.alura.mscompanhiasaereas.infra.mongodb;
```

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.MongoDatabaseFactory;
import org.springframework.data.mongodb.MongoTransactionManager;
import
org.springframework.data.mongodb.config.AbstractMongoClientConfiguration;

@Configuration
class MongoConfiguration extends AbstractMongoClientConfiguration {

    @Value("${spring.data.mongodb.uri}")
    private String mongodbUri;

    @Value("${spring.data.mongodb.database}")
    private String databaseName;

    @Bean
    public MongoTransactionManager transactionManager(MongoDatabaseFactory
dbFactory) {
        return new MongoTransactionManager(dbFactory);
    }

    @Override
    public MongoClient mongoClient() {
        var mongoClientSettings = MongoClientSettings.builder()
                .applyConnectionString(new ConnectionString(mongodbUri))
                .build();
        return MongoClients.create(mongoClientSettings);
    }

    @Override
    protected String getDatabaseName() {
        return databaseName;
```

```
    }
}
```

Agora a anotação `@Transactional` passará a funcionar sem problemas. Podemos inclusive escrever testes automatizados que verifiquem esse comportamento. Para isso, podemos escrever cenários onde:

- a persistência de registros em ambas as *collections* funcione sem problemas;
- um erro ocorra na persistência de uma origem para relatório, fazendo com que o novo voo também não seja cadastrado;
- um erro ocorra na persistência do novo voo, fazendo com que a origem de relatório que deveria ser persistida não seja.

O primeiro cenário pode ser escrito com um teste semelhante ao seguinte. Nele apenas inserimos ambos os registros e verificamos se eles se encontram na base MongoDB:

```
@Test
@Transactional
void cadastrarNovoVoo(){
    vooService.cadastrarNovo(voo);

    assertThat(mongoTemplate.findAll(Voo.class)).contains(voo);

assertThat(mongoTemplate.findAll(OrigemParaRelatorio.class)).contains(orig
emParaRelatorio);
}
```

> A anotação `@Transactional` foi colocada para que os registros recém-inseridos sejam removidos do banco de dados assim que o teste encerrar. Dessa forma, fazemos com que o banco de dados volte ao estado anterior à execução do teste.

Para testarmos o segundo cenário, podemos fazer com que uma exceção seja lançada quando se tentar persistir uma origem para

relatório. Conseguimos esse comportamento anotando o `RepositorioDeOrigensParaRelatorio` com a `@SpyBean`, do Spring.

```
@SpyBean
private RepositorioDeOrigensParaRelatorio
repositorioDeOrigensParaRelatorio;
```

E o teste pode usar o método `doThrow()` do Mockito. A exceção `MongoTransactionException` será lançada assim que o método `save()` de `repositorioOrigensParaRelatorio` for chamado:

```
doThrow(MongoTransactionException.class).when(repositorioDeOrigensParaRela
torio).save(origemParaRelatorio);
```

Já o terceiro cenário aproveita-se da mesma técnica usada no segundo, mas o teste deve lançar uma `MongoTransactionException` quando o método `save()` de `repositorioDeVoos` for chamado:

```
doThrow(MongoTransactionException.class).when(repositorioDeVoos).save(voo)
;
```

O teste completo encontra-se a seguir:

```
package br.com.alura.mscompanhiasaereas.service;

import br.com.alura.mscompanhiasaereas.dominio.RepositorioDeVoos;
import br.com.alura.mscompanhiasaereas.dominio.Voo;
import br.com.alura.mscompanhiasaereas.relatorio.OrigemParaRelatorio;
import
br.com.alura.mscompanhiasaereas.relatorio.RepositorioDeOrigensParaRelatori
o;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.SpyBean;
import org.springframework.data.mongodb.MongoTransactionException;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.transaction.annotation.Transactional;
```

```
import static
br.com.alura.mscompanhiasaereas.dominio.ObjetosParaTestes.doRioParaBeloHor
izonteAmanha;
import static
br.com.alura.mscompanhiasaereas.relatorio.OrigemParaRelatorio.origemDoVoo;
import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.fail;
import static org.mockito.Mockito.doThrow;

@SpringBootTest
class VooServiceIntegrationTest {

    @Autowired
    private VooService vooService;

    @Autowired
    private MongoTemplate mongoTemplate;

    @SpyBean
    private RepositorioDeVoos repositorioDeVoos;

    @SpyBean
    private RepositorioDeOrigensParaRelatorio
repositorioDeOrigensParaRelatorio;

    private Voo voo;
    private OrigemParaRelatorio origemParaRelatorio;

    @BeforeEach
    void setUp(){
        voo = doRioParaBeloHorizonteAmanha();
        origemParaRelatorio = origemDoVoo(voo);
    }

    @Test
    @Transactional
    void cadastrarNovoVoo(){
        vooService.cadastrarNovo(voo);
```

```
        assertThat(mongoTemplate.findAll(Voo.class)).contains(voo);

assertThat(mongoTemplate.findAll(OrigemParaRelatorio.class)).contains(orig
emParaRelatorio);
    }

    @Test
    void erroNoCadastroDeOrigensParaRelatorioImpedeOCadastroDoNovoVoo(){

doThrow(MongoTransactionException.class).when(repositorioDeOrigensParaRela
torio).save(origemParaRelatorio);

        try{
            vooService.cadastrarNovo(voo);
            fail("Deveria ter lançado exception!");
        } catch(MongoTransactionException e){

assertThat(mongoTemplate.findAll(Voo.class)).doesNotContain(voo);

assertThat(mongoTemplate.findAll(OrigemParaRelatorio.class)).doesNotContai
n(origemParaRelatorio);
        }
    }

    @Test
    void erroNoCadastroDoNovoVooImpedeOCadastroDeOrigemParaRelatorio(){

doThrow(MongoTransactionException.class).when(repositorioDeVoos).save(voo)
;
        try{
            vooService.cadastrarNovo(voo);
            fail("Deveria ter lançado exception!");
        } catch(MongoTransactionException e){

assertThat(mongoTemplate.findAll(Voo.class)).doesNotContain(voo);

assertThat(mongoTemplate.findAll(OrigemParaRelatorio.class)).doesNotContai
n(origemParaRelatorio);
        }
```

```
    }
}
```

13.2 Transações de banco de dados e garantias de envio de mensagens

Durante praticamente todo o livro, vimos diversos exemplos de utilização do Kafka Connect. Houve exemplos de conexão com diversos bancos de dados, bem como sua contribuição para a criação de soluções mais resilientes.

Vimos no capítulo 4 alguns conceitos sobre CDC (*Change Data Capture*) e que diversos conectores o implementam. Basicamente, a implementação desse padrão nas aplicações faz com que elas reajam a mudanças feitas em tabelas de banco de dados, sejam inserções, atualizações ou remoções de dados, ou mesmo outras fontes de dados que não bancos. E, a cada uma dessas operações, outros sistemas podem receber os dados envolvidos.

Conforme o livro *Monolith to Microservices* (2019), esse padrão possui grande utilidade quando é necessária uma replicação de dados de uma base de dados para outro sistema. Nesse livro, são vistos exemplos de replicações entre bases de dados de monolitos e de microsserviços recém-extraídos do monolito, usando desde *triggers* de banco de dados até ferramentas que leiam logs de banco, como o Kafka Connect.

Além do CDC, vimos também no capítulo 4 diversos conceitos relacionados à resiliência das aplicações, e como o Kafka Connect nos ajuda com isso.

Mas, embora o Kafka Connect nos ofereça garantias de que os registros enviados para a tabela que ele está escutando serão enviados para o Kafka (seja de forma *online* ou tardia), ainda há um problema a ser resolvido: por diversas razões precisamos inserir,

atualizar ou mesmo remover diversos dados em diferentes tabelas (ou *collections*). E para que essa operação seja atômica, é necessária a utilização de transações. Assim, ou todas as operações em banco são realizadas, ou nenhuma é realizada.

Repare que, devido ao contexto transacional, também temos garantias de que o envio da mensagem ao Kafka será controlado pela transação de banco. Tomando como exemplo as *collections* usadas neste capítulo:

- `origens_relatorio`
- `voos`

Se por algum motivo a persistência ocorrer em apenas uma das *collections* acima, a mensagem não será enviada ao Kafka. Isso ajuda bastante, no sentido de não precisarmos lidar com transações distribuídas tradicionais, também conhecidas como 2PC (*two phase commit*):

- uma envolvendo o banco de dados
- outra envolvendo o envio da mensagem ao Kafka. Até porque o Kafka não suporta transações distribuídas

Além disso, é possível notar que o produtor da mensagem não tem nem mesmo acesso ao Kafka. Tudo o que ele precisa fazer é persistir de forma transacional um registro. O Kafka Connect cuidará do resto.

Essa solução para produção de mensagens com garantia de envio é também um padrão para microsserviços catalogado no livro *Microsservices Patterns* (2019): o **Transactional Outbox**. E veremos mais sobre ele a seguir.

13.3 O Transactional Outbox Pattern e os padrões Polling Publisher e Transaction Log

Tailing

Para o padrão *Transactional Outbox*, a tabela (ou *collection*) `outbox` é como se fosse uma caixa de saída de email. Todas as informações relacionadas à mensagem devem estar nessa tabela. No exemplo deste capítulo, podemos considerar a *collection* `voos` como sendo nossa caixa de saída, visto que é ela que é lida pelo Kafka Connect.

São necessários também outros participantes:

- o serviço que envia a mensagem (representado pelo `VooService`);
- o banco de dados (o MongoDB, no nosso caso);
- a caixa de saída (representada pela *collection* `voos`);
- um *message relay*, que lê a fonte de dados e envia as alterações dos dados ao destino (representado pelo Kafka Connect).

A figura a seguir mostra a estrutura do padrão, aplicada ao *ms-companhias-aereas*.

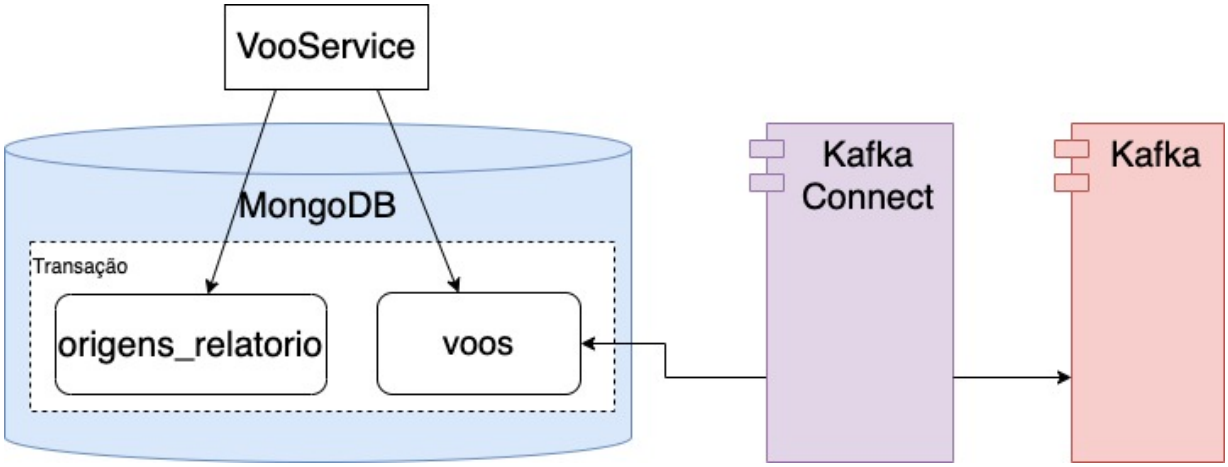


Figura 13.1: Estrutura do padrão Transactional Outbox.

Talvez a única diferença entre a nossa implementação e o padrão catalogado por Richardson é que não utilizamos uma *collection* exclusiva para o envio da mensagem, mas uma que pertença a aplicação.

É possível também determinar de que maneira o *message relay* lerá as mensagens. Para isso temos os padrões *Polling Publisher* e o *Transactional Log Tailing*:

- no *Polling Publisher*, o *message relay* acessa a caixa de saída, de tempos em tempos, em busca de alterações que não foram enviadas, e as envia para o destino. Pode-se usar o *cron* ou mesmo ferramentas como Quartz para agendamento dessa tarefa
- já no *Transaction Log Tailing*, por meio da leitura do log do banco de dados, busca-se as alterações na caixa de saída que ainda não foram enviadas. Esse é o padrão aplicado pelo Kafka Connect. Ou seja, parte-se do princípio de que todas as modificações realizadas no banco estarão nesse log. E o Kafka Connect seleciona apenas aquelas que foram realizadas na caixa de saída

Algumas vantagens de se trabalhar com o *Transaction Log Tailing* em detrimento do *Polling Publisher*:

- o *Polling Publisher* implica em leituras constantes da caixa de saída, o que pode ser algo caro, dependendo da quantidade de escritas e leituras realizadas no banco, seja pelo *message relay*, seja pela aplicação que utiliza o banco;
- ao usar o *Polling Publisher*, é provável que sejam necessárias colunas adjacentes às já existentes, para controle de mensagens já enviadas, como colunas que representem a data de persistência do registro na caixa de saída por exemplo. Assim, a *query* de busca pode se basear nelas para determinar quais registros ainda não foram lidos;
- o *Polling Publisher* não obriga a utilização de uma caixa de saída exclusiva, podendo usar qualquer tabela ou *collection* como tal. Entretanto, podem ocorrer os problemas citados no primeiro *bullet*;
- o *Transaction Log Tailing* realiza leituras no log do banco de dados, não implicando em leituras diretas na caixa de saída. Dessa forma, o banco de dados não tem sua performance afetada.

13.4 Resumo

Neste apêndice, revisamos a importância do Kafka Connect na implementação do padrão CDC, além de mostrar o padrão *Transaction Outbox* e padrões necessários à sua implementação, como o *Polling Publisher* e o *Transaction Log Tailing*.

E com isso, encerramos este livro. Espero que você tenha gostado da leitura.

O Kafka Connect é uma ferramenta que pode ser usada tanto na conexão entre aplicações da forma mais desacoplada possível (aproveitando-se também das vantagens da comunicação assíncrona entre aplicações) quanto como auxiliar na extração de microsserviços em monolitos. Também pode ser usada para disponibilizar dados de sistemas legados (que podem não suportar uma conexão direta com o Kafka), entre outras utilidades.

Vale a pena aprender a ferramenta, e quem sabe, utilizá-la na sua próxima solução distribuída.

Capítulo 14
Bibliografia

BLOCH, J. *Effective Java*. 3rd ed. edição ed. Boston: Addison-Wesley Professional, 2018.

FOWLER, M. *Refactoring*: Improving the Design of Existing Code. 2nd edition ed. Boston: Addison-Wesley Professional, 2018.

HOHPE, G.; WOOLF, B. *Enterprise Integration Patterns*: Designing, Building, and Deploying Messaging Solutions. 1st edition ed. Boston Munich: Addison-Wesley Professional, 2003.

MARTIN, R. C.; FEATHERS, M. C.; OTTINGER, T. R. *Clean Code*: A Handbook of Agile Software Craftsmanship. Illustrated edição ed. Upper Saddle River, NJ Munich: Prentice Hall PTR, 2008.

MCCONNELL, S. *Code Complete*: A Practical Handbook of Software Construction. 2nd ed. edição ed. Redmond, Washington: Microsoft Press, 2004.

Michaelis On-Line. Disponível em: https://michaelis.uol.com.br/. Acesso em: 8 set. 2023.

NEWMAN, S. *Building Microservices*. 2. ed. United States: O'Reilly Media, Inc., 2021.

NEWMAN, S. *Monolith to Microservices*. United States: O'Reilly Media, Inc., 2019.

NYGARD, M. T. *Release It!* Design and Deploy Production-Ready Software. 1. ed. United States: O'Reilly Media, Inc., 2007.

RICHARDSON, C. *Microservices Patterns*. 1. ed. United States: Manning, 2019.

SANTANA, E. F. Z. *Apache Kafka e Spring Boot*: Comunicação Assíncrona Entre Microsserviços. 1. ed. Brasil: Casa do Código, 2022.

SHAPIRA, G.; NARKHEDE, N.; PALINO, T. *Kafka - The Definitive Guide*: Real-Time Data and Stream Processing at Scale. 1ª edição ed. Sebastopol, CA: O'Reilly Media, Inc, USA, 2017.

SILVEIRA, P.; TURINI, R. *Java 8 Prático*: Lambdas, Streams e os Novos Recursos da Linguagem. 1. ed.

Brasil: Casa do Código, 2014.

THOMAS, D.; HUNT, A. *The Pragmatic Programmer*: Your Journey To Mastery, 20th Anniversary Edition. 2nd edition ed. Boston: Addison-Wesley Professional, 2019.